**AI-LAB REPORT**

**IMPLEMENT TIC TAC TOE: DATE:1-10-24**

#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:

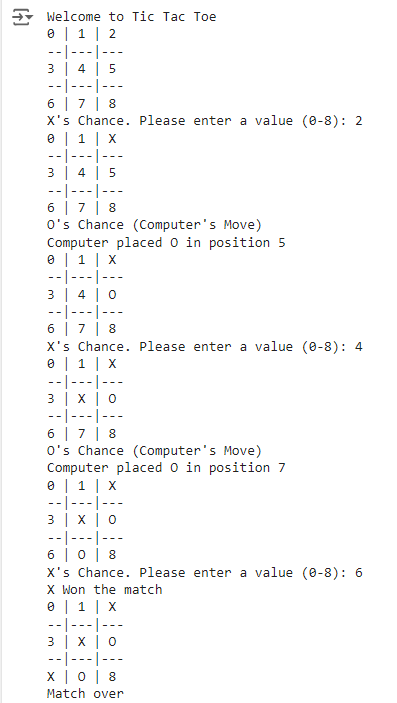
            printBoard(xState, zState)

            print("Match over")

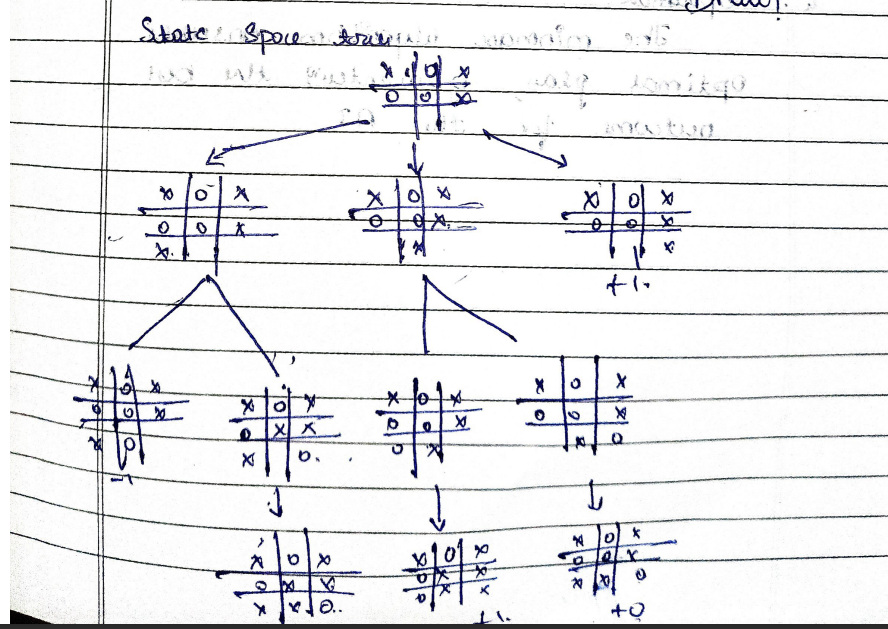
            break

        turn = 1 - turn  # Switch turn

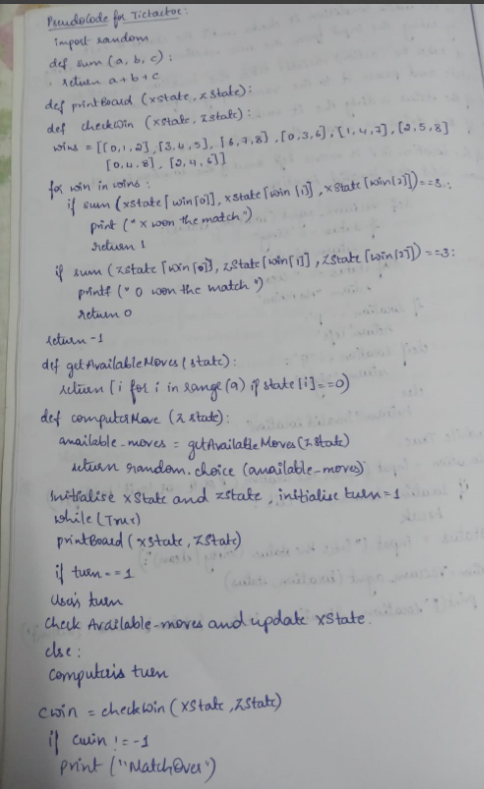
**OUTPUT:**



**STAE SPACE TREE:**



**ALGORITHM OR PSEUDO CODE :**

****

**IMPEMENT VACCUME AGENT PROGRAM: DATE:1-10-24**

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':

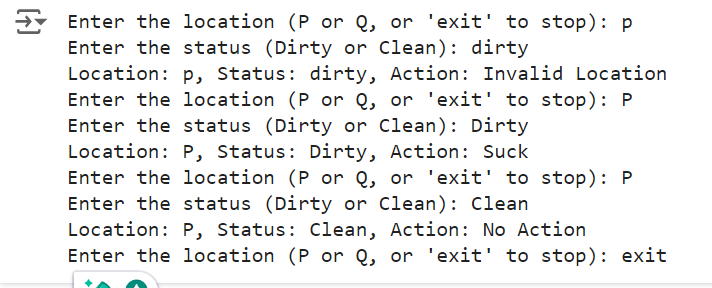
        break

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

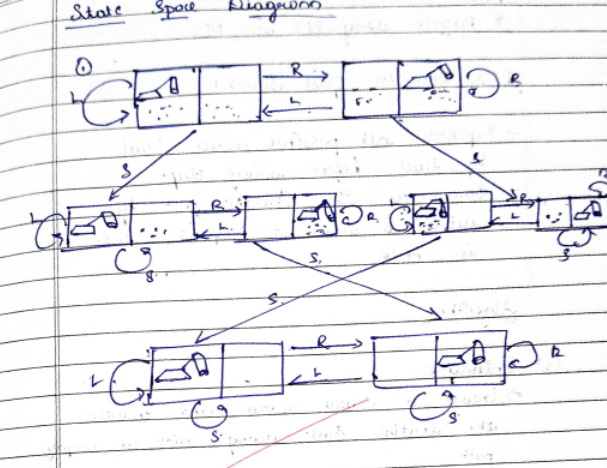
    action = vacuum\_agent(location, status)

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

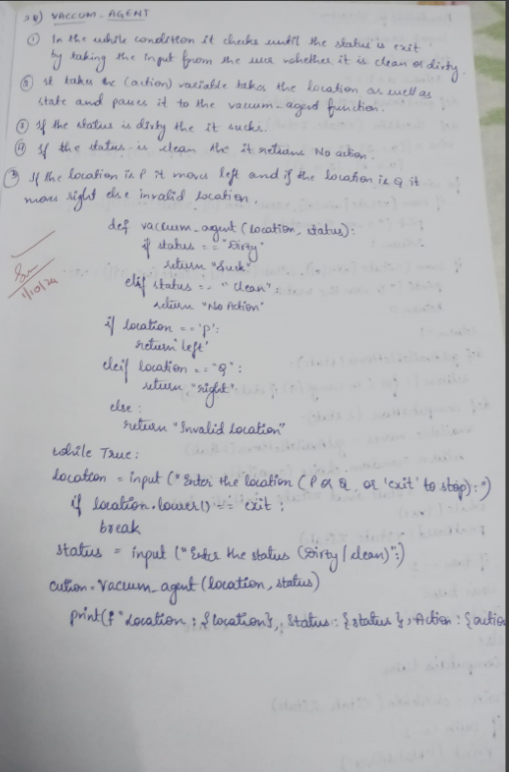
**OUTPUT:**

****

**STATE SPACE TREE:**

****

**ALGORITHM OR PSEUDO CODE:**

****

**SOLVE 8 PUZZLE PROBLEM USING DFS: DATE:15-10-24**

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)

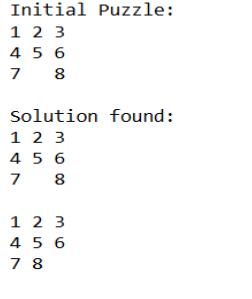
    if not dfs(root):

        print("Failed to find a solution.")

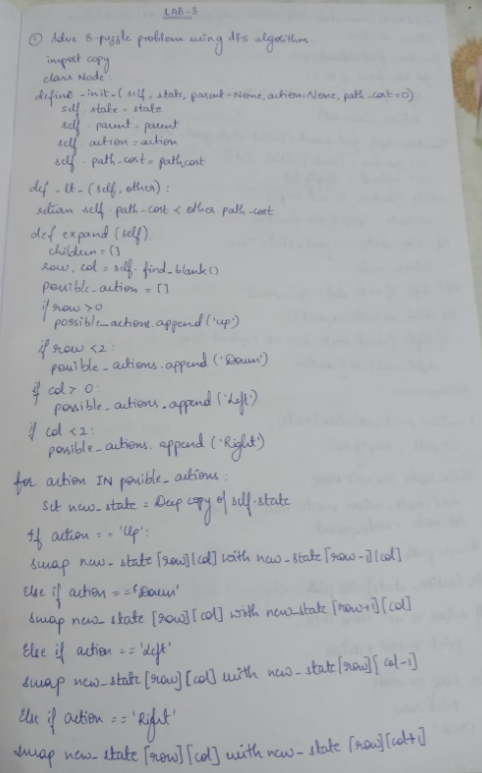
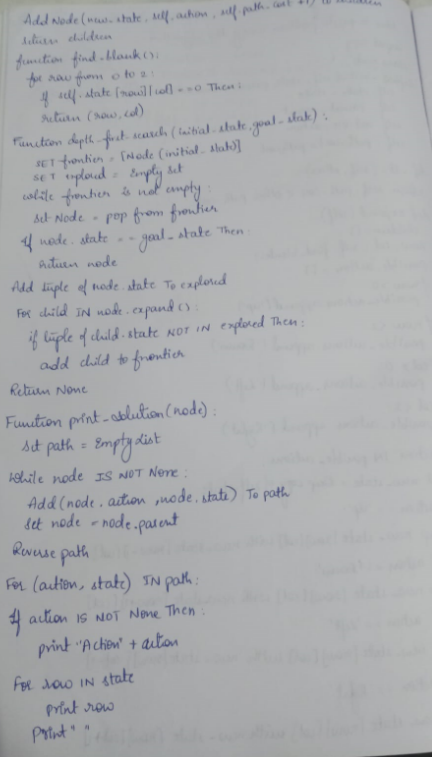
if \_\_name\_\_ == "\_\_main\_\_":

    main()

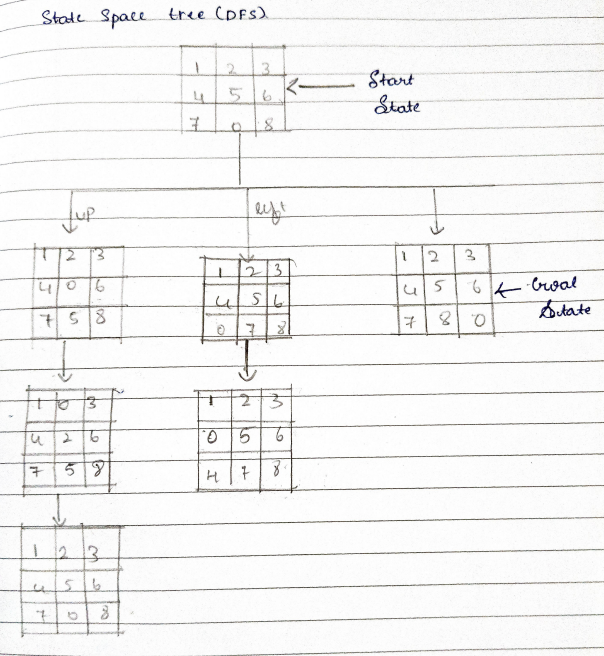
**OUTPUT:**



**ALGORITHM OR PSEUDO CODE:**

**** ****

**STATE SPACE TREE:**



**ITERATIVE DEEPENING DFS: DATE:12-10-24**

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)

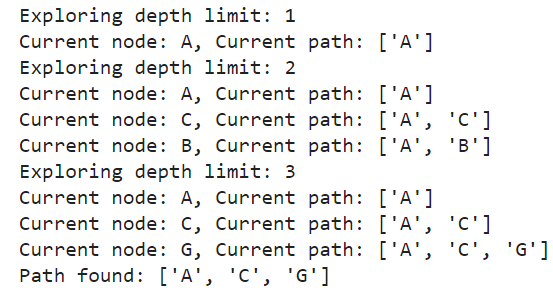
if path:

    print("Path found:", path)

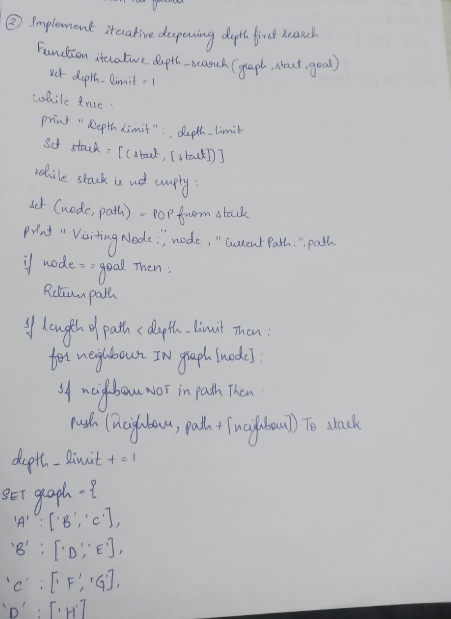
else:

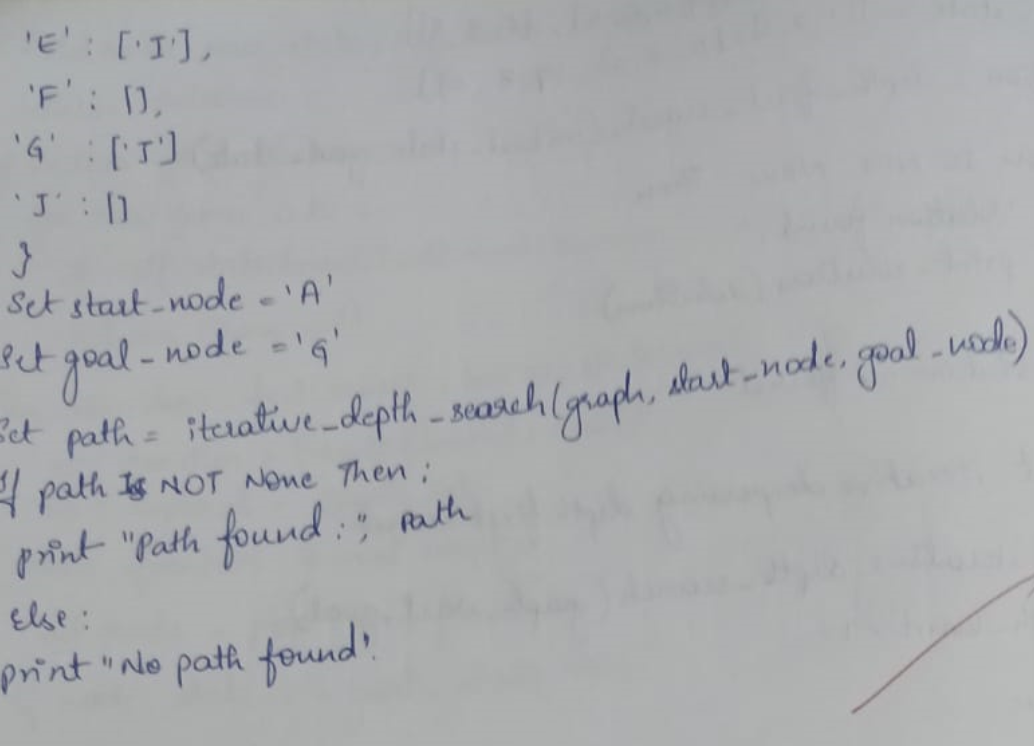
    print("No path found.")

**OUTPUT:**

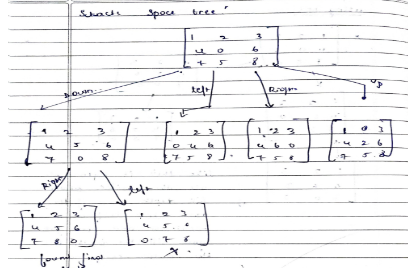
****

**ALGORITHM OR PSEUDO CODE:**

****

****

**STATE SPACE TREE:**

****

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

**DATE:12-10-24**

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)

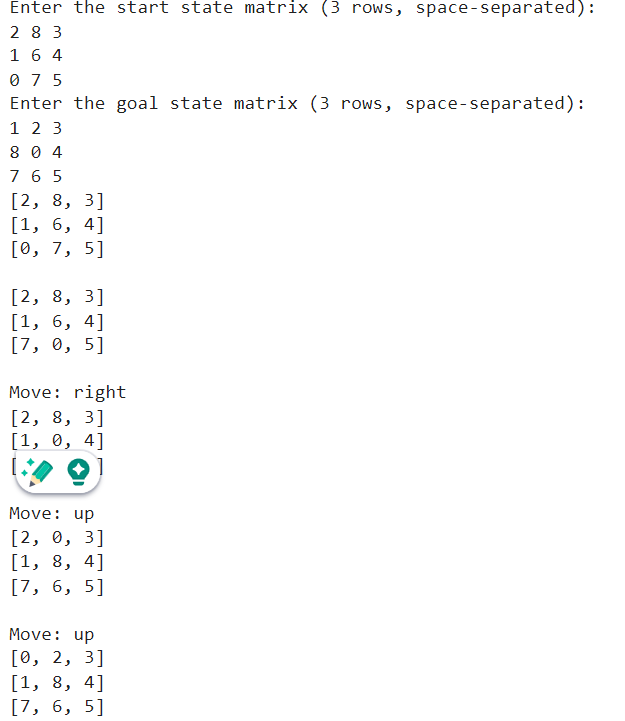
    else:

        print("No solution found.")

except Exception as e:

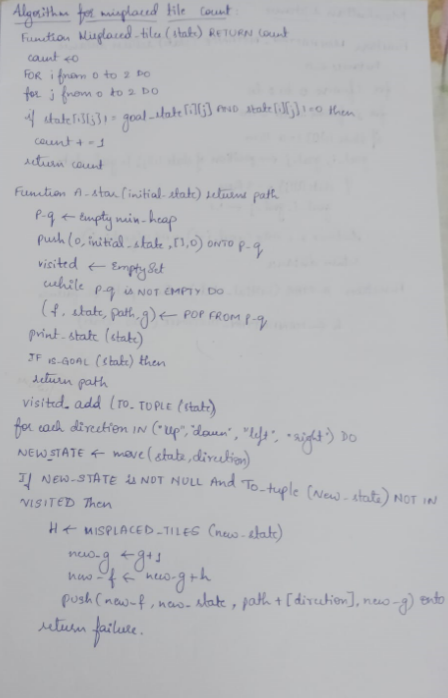
    print("Error:", str(e))

**OUTPUT:**

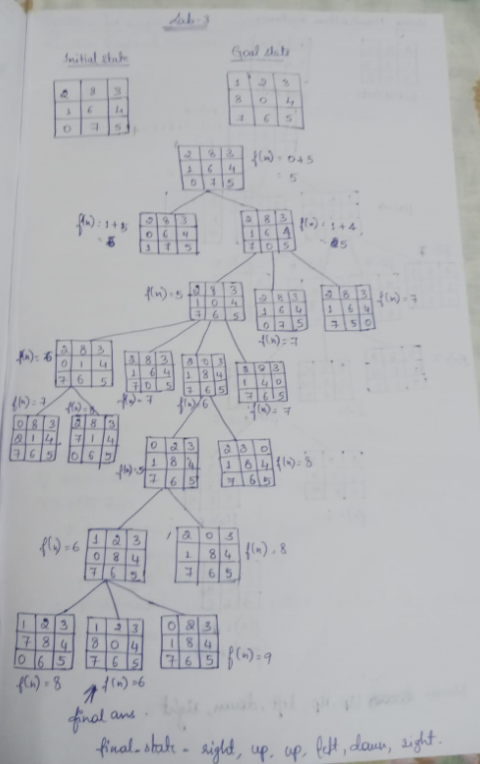
****

****

**ALGORITHM OR PSEUDO CODE:**

****

**STATE SPACE TREE:**

****

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

**DATE:12-10-24**

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:

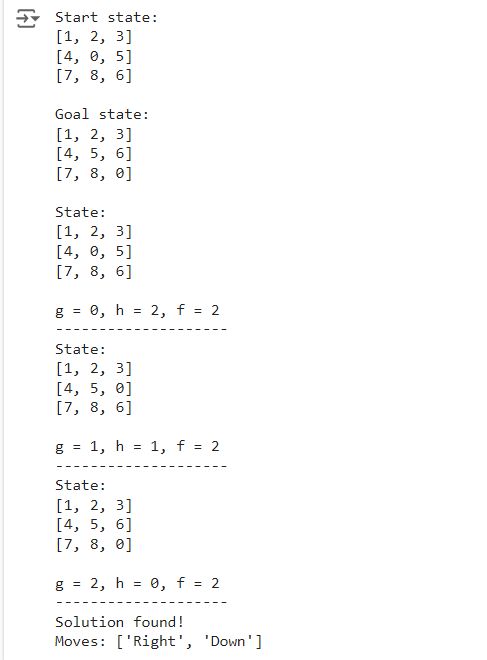
        print("Solution found!")

        print("Moves:", solution)

    else:

        print("No solution found.")

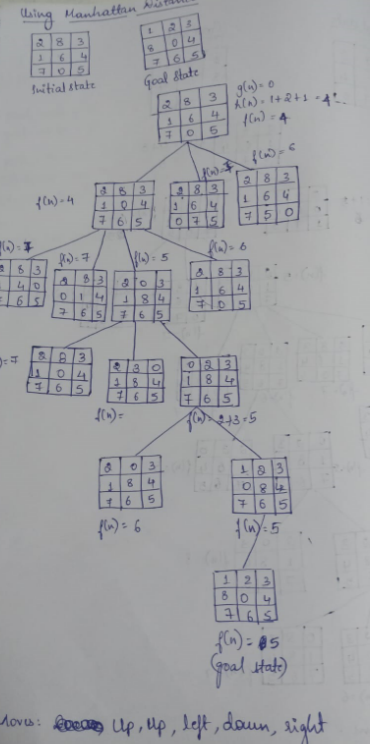
**OUTPUT:**

****

**ALGORITHM OR PSEUDO CODE:**

****

**STATE SPACE TREE:**

****

# IMPLEMENT HILL CLIMBING ALGORITHM DATE:

# def calculate\_heuristic(state):

# heuristic = 0

# n = len(state)

# for i in range(n):

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

# if state[i] == state[j]:  # Same column

# heuristic += 1

# if abs(state[i] - state[j]) == abs(i - j):  # Same diagonal

# heuristic += 1

# return heuristic

# # Function to generate neighboring states by swapping

# def generate\_neighbors(state):

# neighbors = []

# n = len(state)

# # Generate neighbors by swapping positions of two queens

# for i in range(n):

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

# new\_state = state.copy()

# new\_state[i], new\_state[j] = new\_state[j], new\_state[i]  # Swap

# neighbors.append(new\_state)

# return neighbors

# # Function to print the board as a 2D array

# def print\_board(state):

# n = len(state)

# board = [['.'] \* n for \_ in range(n)]

# for row in range(n):

# board[row][state[row]] = 'Q'

# for row in board:

# print(' '.join(row))

# print()

# # Hill Climbing algorithm for N-Queens with swapping

# def hill\_climbing\_n\_queens(initial\_state):

# current\_state = initial\_state

# 

# while True:

# current\_heuristic = calculate\_heuristic(current\_state)

# print(f"Current State: {current\_state}, Heuristic: {current\_heuristic}")

# print\_board(current\_state)  # Print the current board

# 

# if current\_heuristic == 0:

# return current\_state

# 

# neighbors = generate\_neighbors(current\_state)

# best\_neighbor = None

# best\_heuristic = float('inf')

# 

# for neighbor in neighbors:

# heuristic = calculate\_heuristic(neighbor)

# if heuristic < best\_heuristic:

# best\_heuristic = heuristic

# best\_neighbor = neighbor

# 

# if best\_heuristic >= current\_heuristic:

# break  # Stop if no better neighbor found

# 

# current\_state = best\_neighbor

# 

# return None  # No solution found

# # Main function to solve the N-Queens problem

# def solve\_n\_queens():

# # Set the initial state for 4 queens

# initial\_state = [3, 1, 2, 0]  # Fixed state

# solution = hill\_climbing\_n\_queens(initial\_state)

# 

# if solution:

# print(f"Solution found for 4-Queens problem: {solution}")

# print\_board(solution)  # Print the final board

# else:

# print("No solution found.")

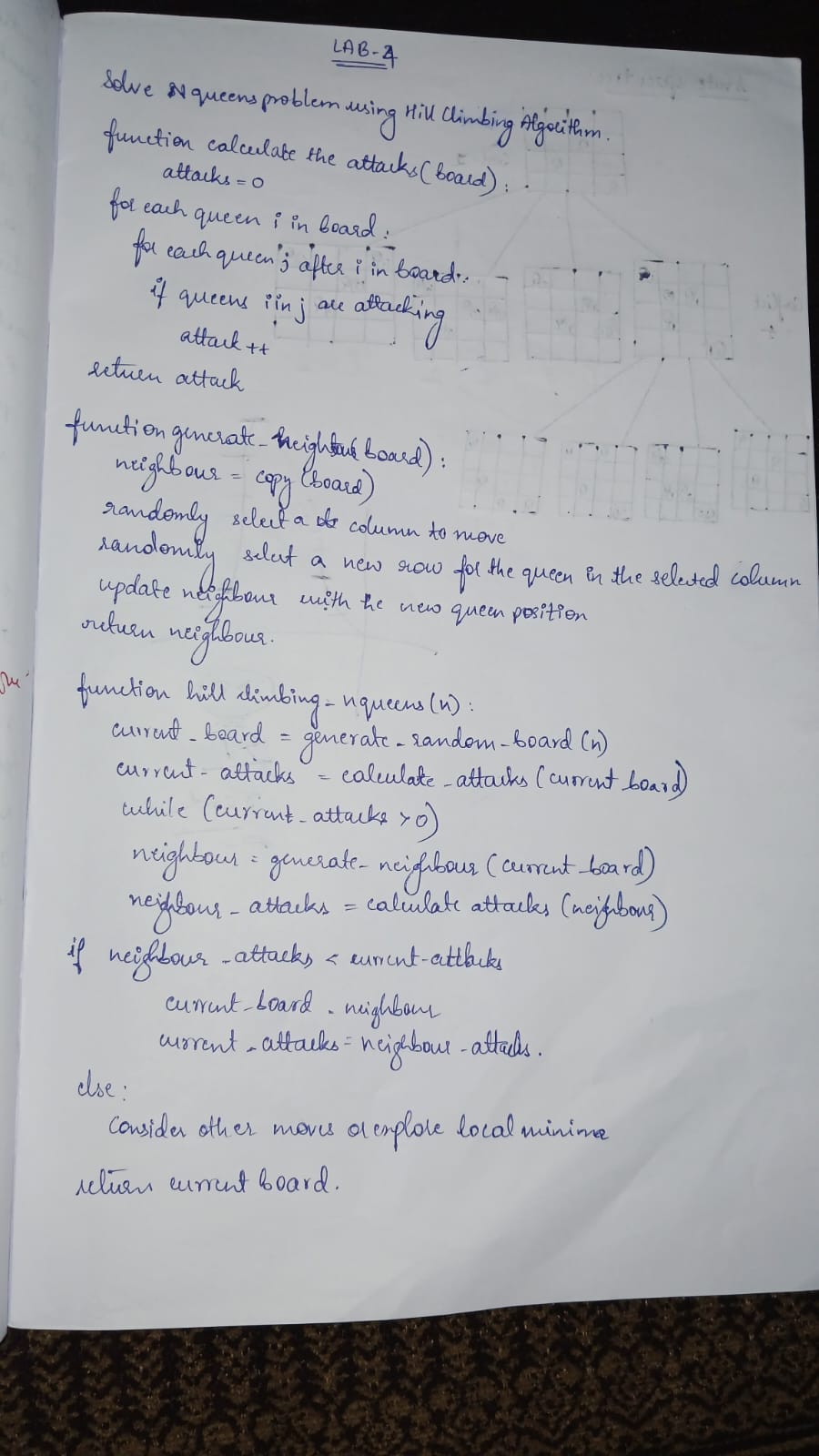
# # Run the solver

# solve\_n\_queens()

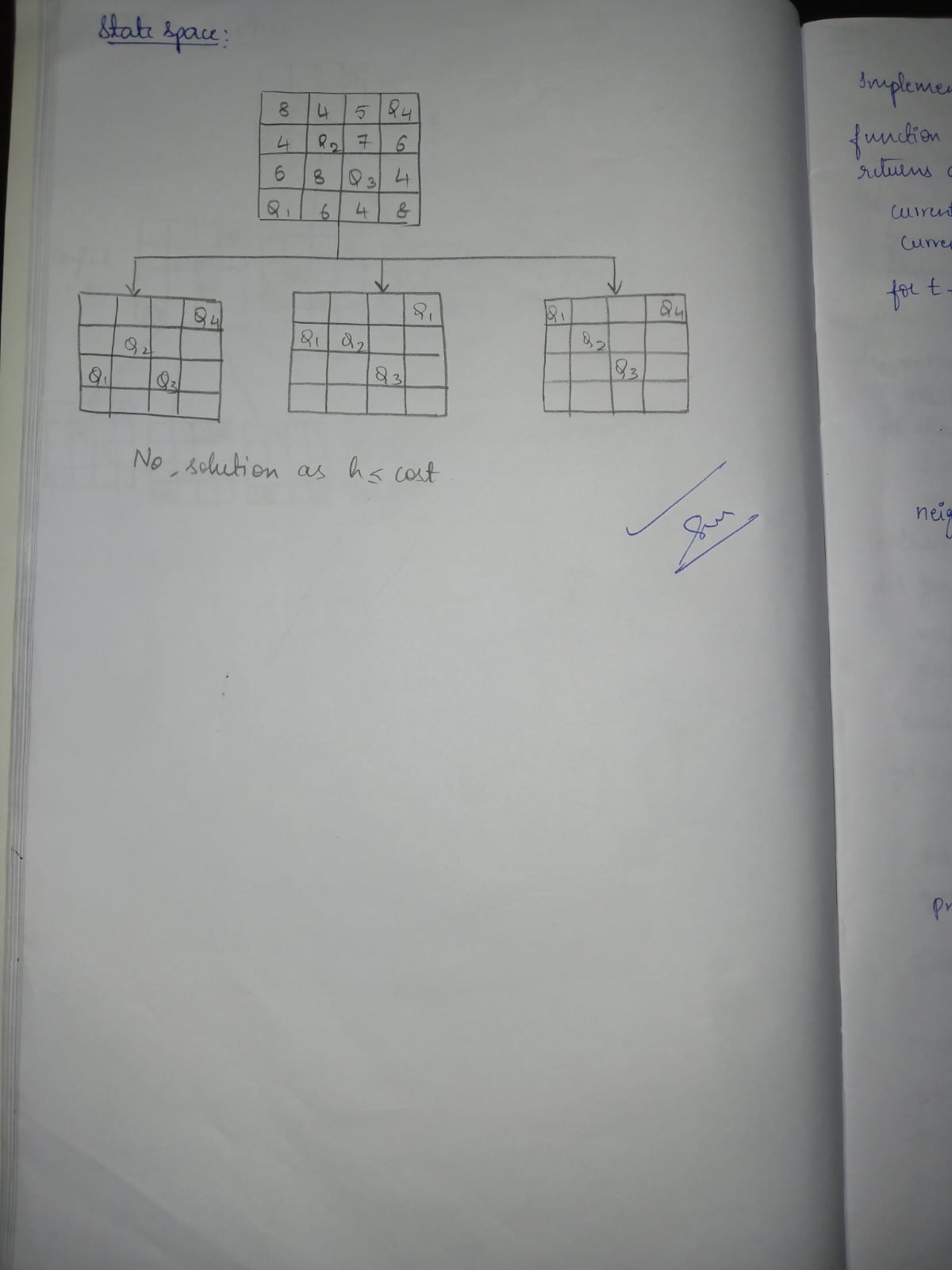
**OUTPUT:**



**ALGORITHM:**



**STATE SPACE:**



**IMPLEMENT SIMULATED ANNEALING ALGORITHM:**

**DATE:12-11-24**

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:

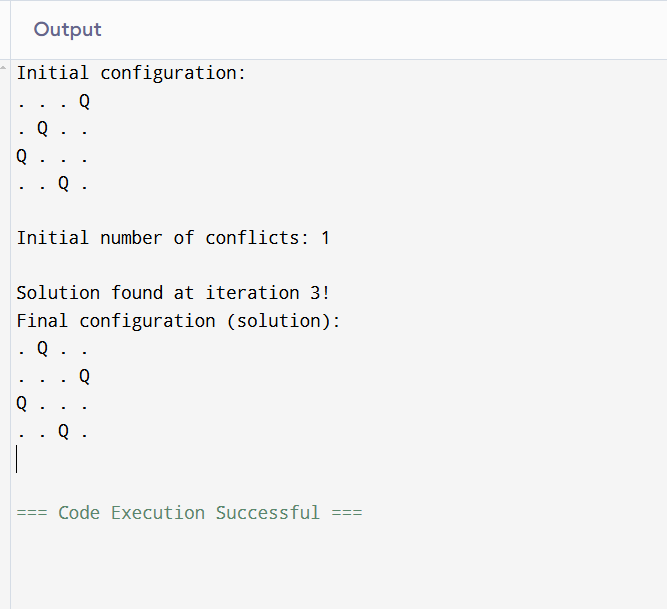
print(f"Solution found: {initial\_state}")

else:

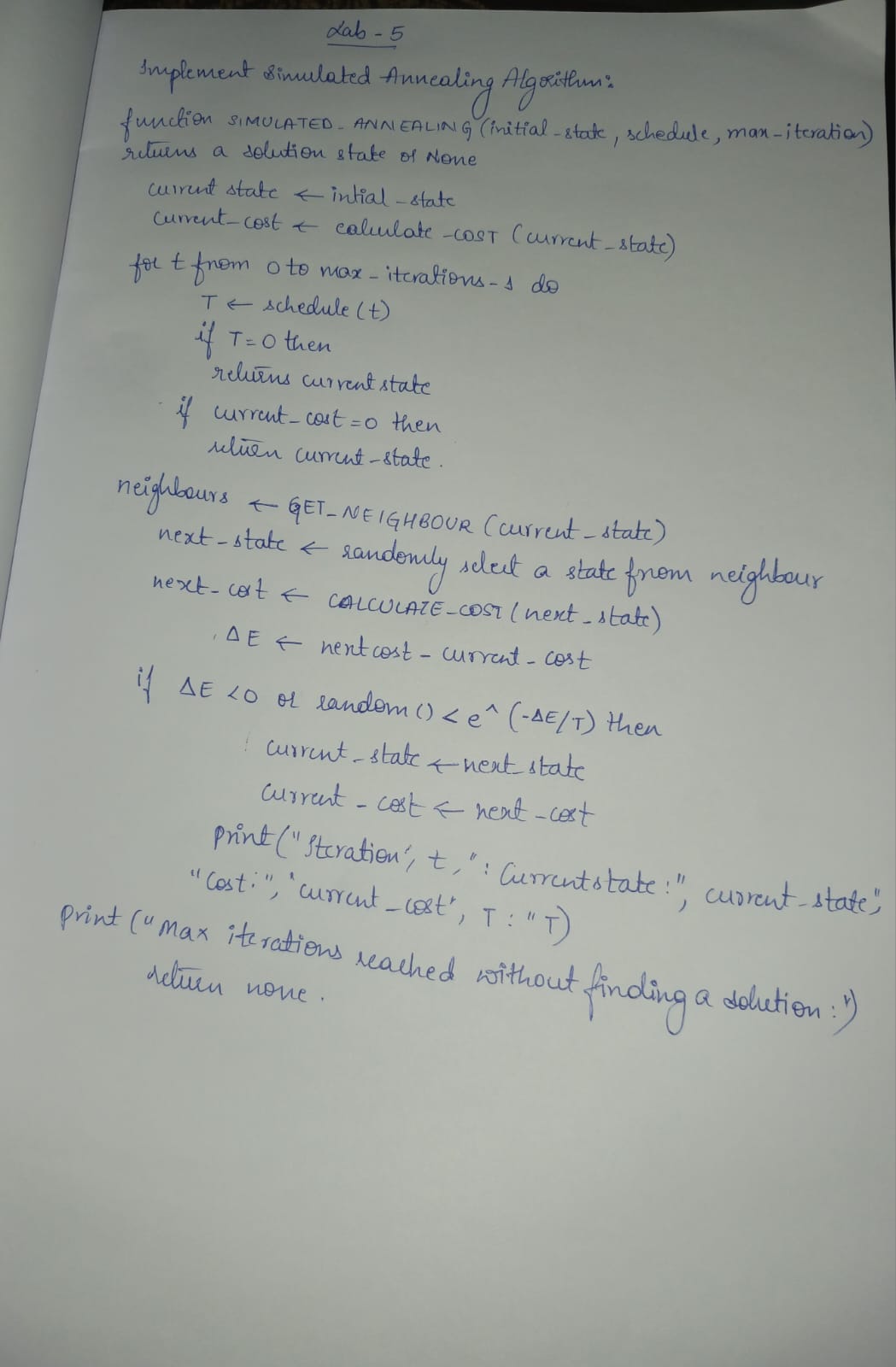
print("Max iterations reached without finding a

solution.")

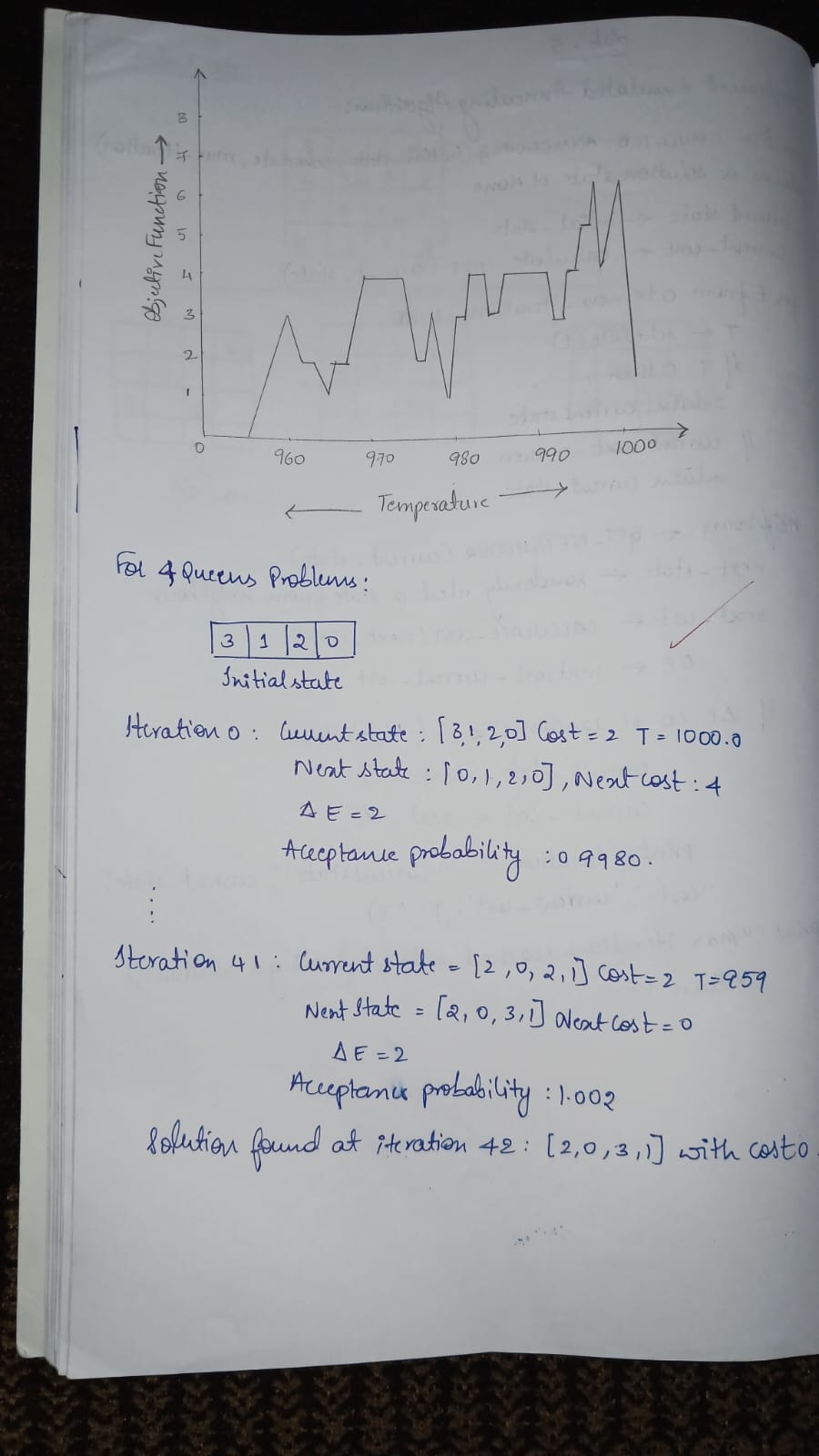
**OUTPUT:**



**ALGORITHM:**

****

**STATE SPACE TREE:**



# PROPOSITIONAL LOGIC

# DATE:12-11-24

# combinations = [(True, True, True), (True, True, False), (True, False, True), (True, False, False),

# (False, True, True), (False, True, False), (False, False, True), (False, False, False)]

# variable = {'p': 0, 'q': 1, 'r': 2}

# kb = ''

# q = ''

# priority = {'~': 3, 'v': 1, '^': 2}

# def input\_rules():

# global kb, q

# kb = input("Enter rule: ")

# q = input("Enter the Query: ")

# def entailment():

# global kb, q

# print('\*' \* 10 + "Truth Table Reference" + '\*' \* 10)

# print('kb', 'alpha')

# print('\*' \* 10)

# for comb in combinations:

# s = evaluatePostfix(toPostfix(kb), comb)

# f = evaluatePostfix(toPostfix(q), comb)

# print(s, f)

# print('-' \* 10)

# if s and not f:

# return False

# return True

# def isOperand(c):

# return c.isalpha() and c != 'v'

# def isLeftParanthesis(c):

# return c == '('

# def isRightParanthesis(c):

# return c == ')'

# def isEmpty(stack):

# return len(stack) == 0

# def peek(stack):

# return stack[-1]

# def hasLessOrEqualPriority(c1, c2):

# try:

# return priority[c1] <= priority[c2]

# except KeyError:

# return False

# def toPostfix(infix):

# stack = []

# postfix = ''

# for c in infix:

# if isOperand(c):

# postfix += c

# else:

# if isLeftParanthesis(c):

# stack.append(c)

# elif isRightParanthesis(c):

# operator = stack.pop()

# while not isLeftParanthesis(operator):

# postfix += operator

# operator = stack.pop()

# else:

# while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):

# postfix += stack.pop()

# stack.append(c)

# while (not isEmpty(stack)):

# postfix += stack.pop()

# return postfix

# def evaluatePostfix(exp, comb):

# stack = []

# for i in exp:

# if isOperand(i):

# stack.append(comb[variable[i]])

# elif i == '~':

# val1 = stack.pop()

# stack.append(not val1)

# else:

# val1 = stack.pop()

# val2 = stack.pop()

# stack.append(\_eval(i, val2, val1))

# return stack.pop()

# def \_eval(i, val1, val2):

# if i == '^':

# return val2 and val1

# return val2 or val1

# input\_rules()

# ans = entailment()

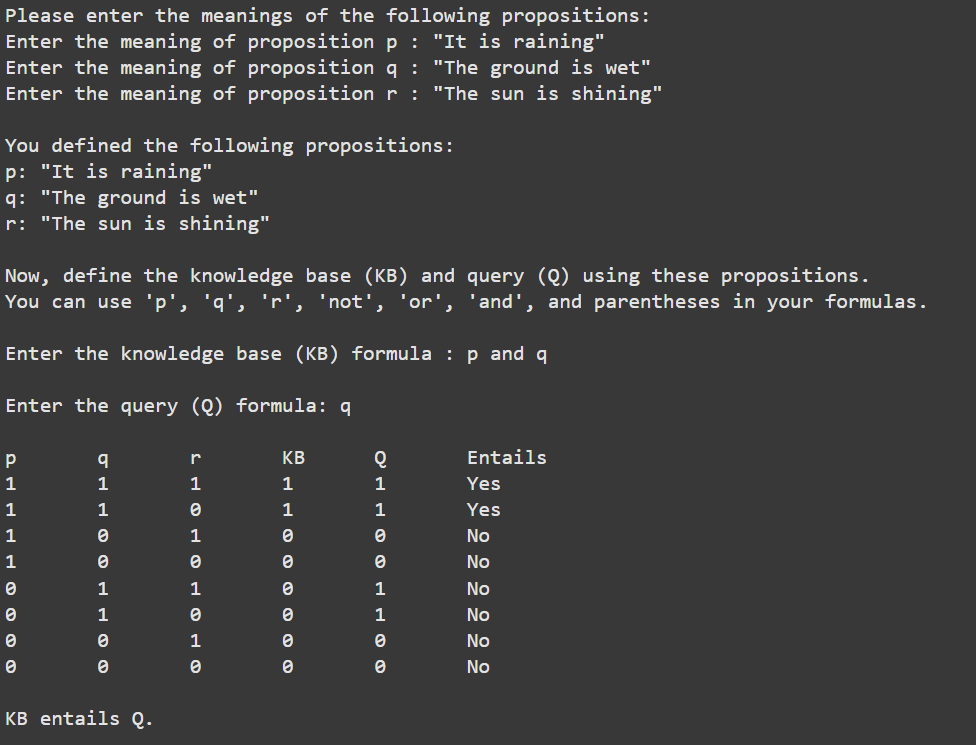
# if ans:

# print("The Knowledge Base entails query")

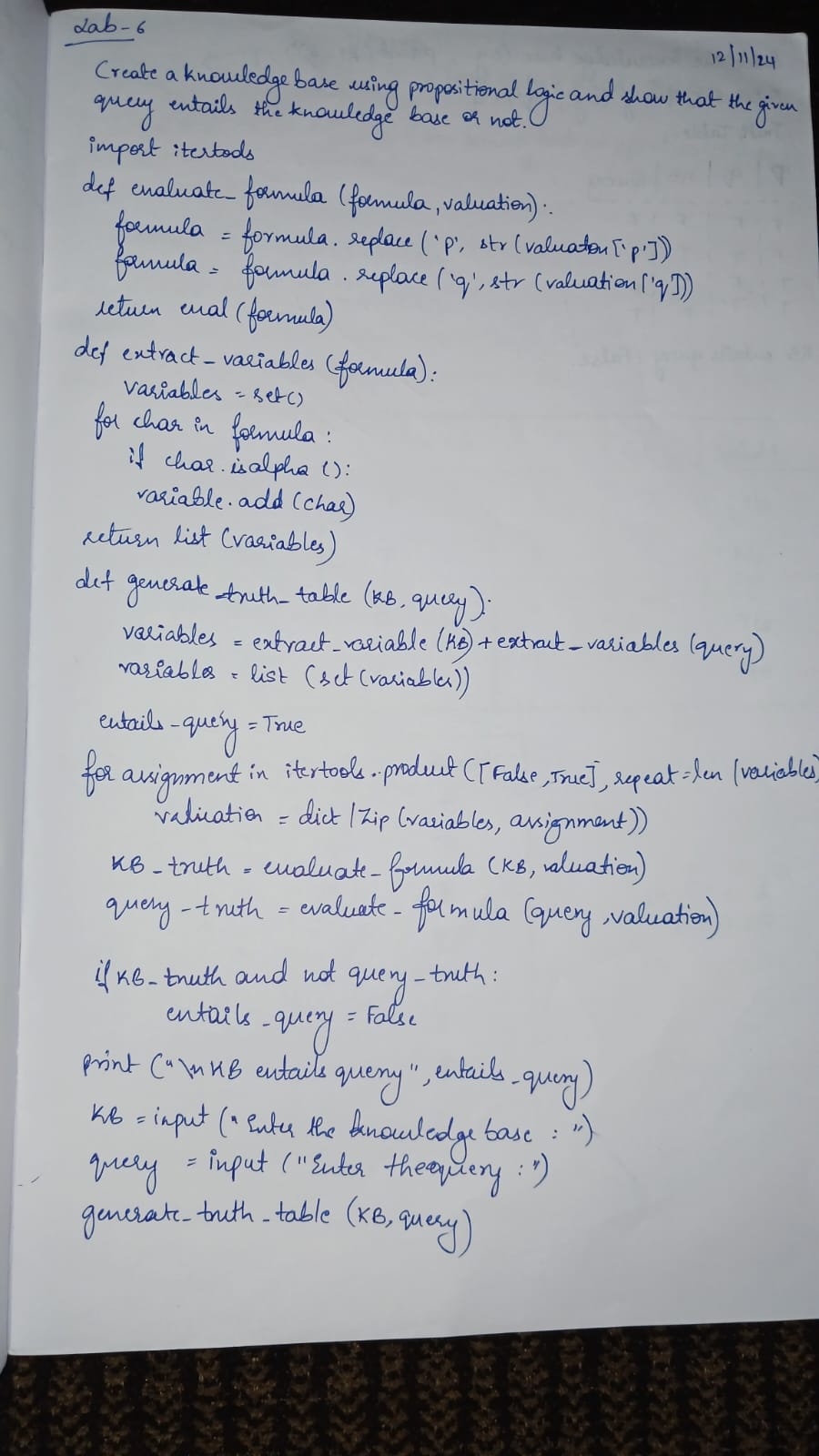
# else:

# print("The Knowledge Base does not entail query")

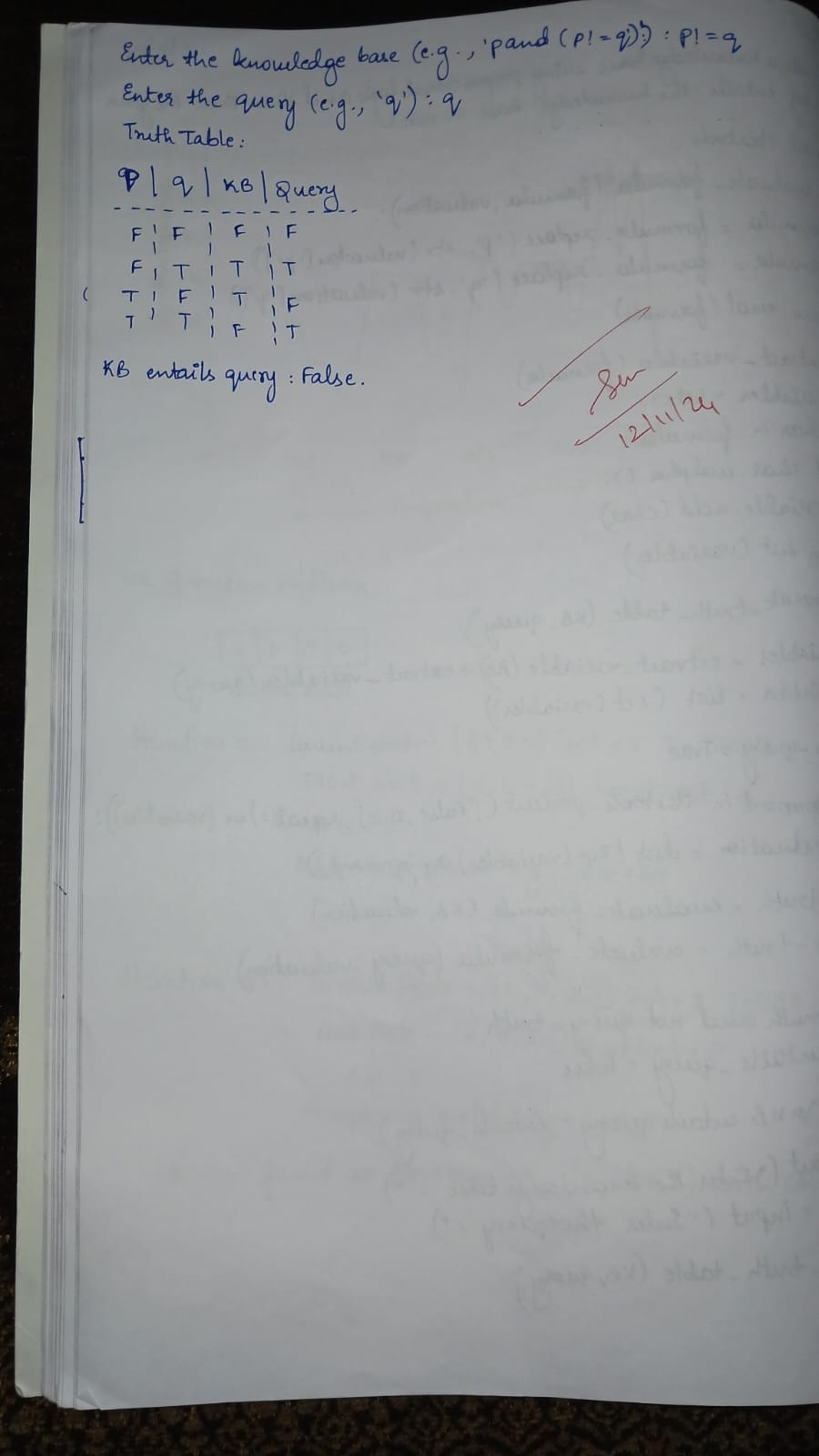
**OUTPUT:**

****

**ALGORITHM:**



**STATE SPACE:**



**FIRST ORDER LOGIC DATE:19-11-24**

def negate(literal):

    """Return the negation of a literal."""

    if isinstance(literal, tuple) and literal[0] == "not":

        return literal[1]

    else:

        return ("not", literal)

def resolve(clause1, clause2):

    """Return the resolvent of two clauses."""

    resolvents = set()

    for literal1 in clause1:

        for literal2 in clause2:

            if literal1 == negate(literal2):

                resolvent = (clause1 - {literal1}) | (clause2 - {literal2})

                print(f"    Resolving literal: {literal1} with {literal2}")

                print(f"    Resulting Resolvent: {resolvent}")

                resolvents.add(frozenset(resolvent))

    return resolvents

def resolution\_algorithm(KB, query):

    """Perform the resolution algorithm to check if the query can be proven."""

    print("\n--- Step-by-Step Resolution Process ---")

    negated\_query = negate(query)

    KB.append(frozenset([negated\_query]))

    print(f"Negated Query Added to KB: {negated\_query}")

    clauses = set(KB)

    step = 1

    while True:

        new\_clauses = set()

        print(f"\nStep {step}: Resolving Clauses")

        for c1 in clauses:

            for c2 in clauses:

                if c1 != c2:

                    print(f"  Resolving clauses: {c1} and {c2}")

                    resolvent = resolve(c1, c2)

                    for res in resolvent:

                        if frozenset([]) in resolvent:

                            print("\nEmpty clause derived! The query is provable.")

                            return True

                        new\_clauses.add(res)

        if new\_clauses.issubset(clauses):

            print("\nNo new clauses can be derived. The query is not provable.")

            return False

        clauses.update(new\_clauses)

        step += 1

KB = [

    frozenset([("not", "food(x)"), ("likes", "John", "x")]),

    frozenset([("food", "Apple")]),

    frozenset([("food", "vegetables")]),

    frozenset([("not", "eats(y, z)"), ("killed", "y"), ("food", "z")]),

    frozenset([("eats", "Anil", "Peanuts")]),

    frozenset([("alive", "Anil")]),

    frozenset([("not", "eats(Anil, w)"), ("eats", "Harry", "w")]),

    frozenset([("killed", "g"), ("alive", "g")]),

    frozenset([("not", "alive(k)"), ("not", "killed(k)")]),

    frozenset([("likes", "John", "Peanuts")])

]

query = ("likes", "John", "Peanuts")

result = resolution\_algorithm(KB, query)

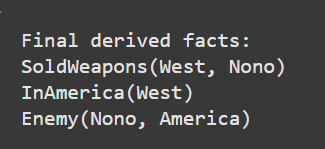
if result:

    print("\nQuery is provable.")

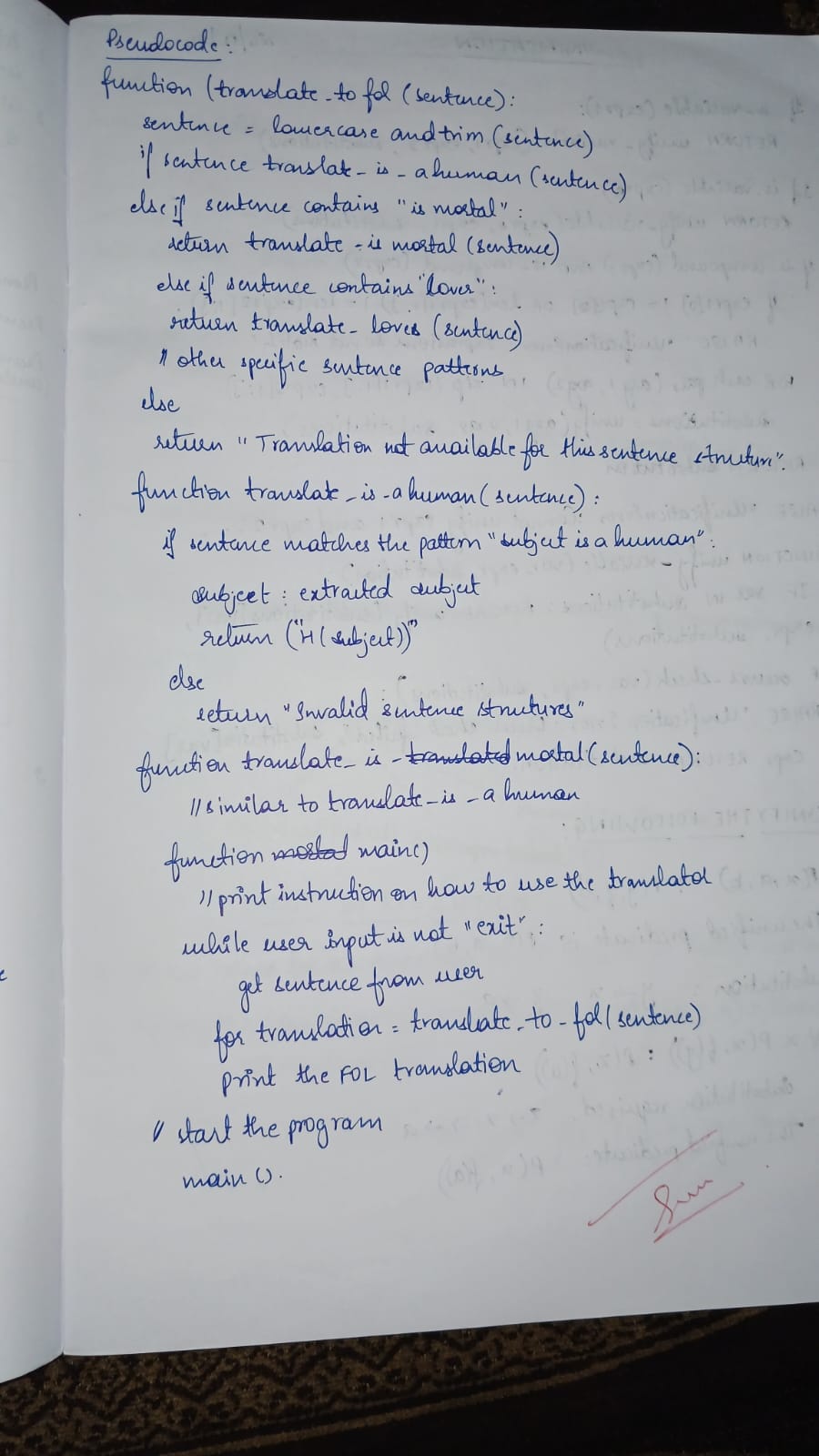
else:

    print("\nQuery is not provable.")

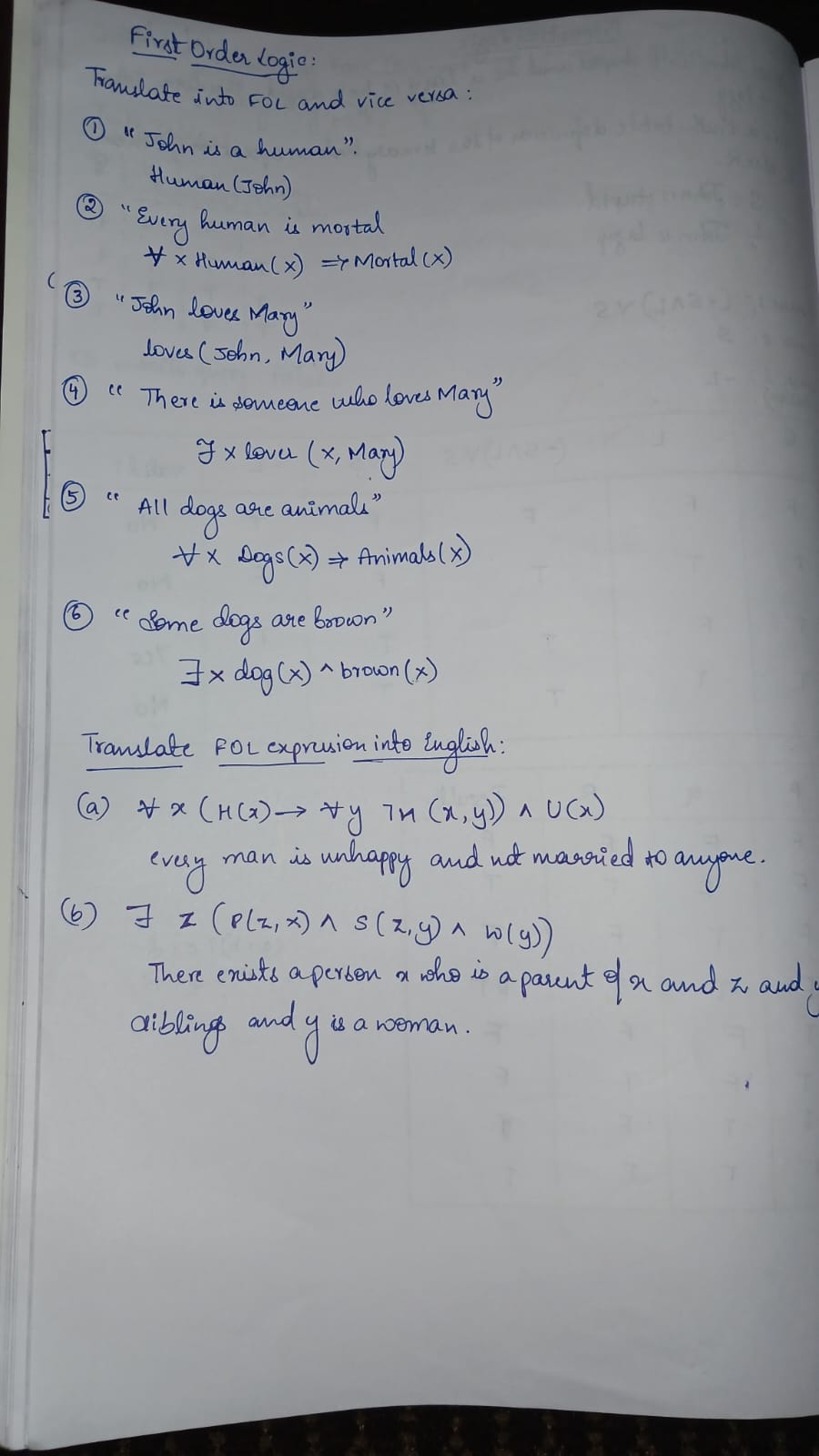
**OUTPUT:**

****

**ALGORITHM:**



**STATE SPACE:**



# UNIFICATION DATE: 26-11-24

# class UnificationError(Exception):

# pass

# def unify(expr1, expr2, substitutions=None):

# if substitutions is None:

# substitutions = {}

# # If both expressions are identical, return current substitutions

# if expr1 == expr2:

# return substitutions

# # If the first expression is a variable

# if is\_variable(expr1):

# return unify\_variable(expr1, expr2, substitutions)

# # If the second expression is a variable

# if is\_variable(expr2):

# return unify\_variable(expr2, expr1, substitutions)

# # If both expressions are compound expressions

# if is\_compound(expr1) and is\_compound(expr2):

# if expr1[0] != expr2[0] or len(expr1[1:]) != len(expr2[1:]):

# raise UnificationError("Expressions do not match.")

# return unify\_lists(expr1[1:], expr2[1:], unify(expr1[0], expr2[0], substitutions))

# # If expressions are not compatible

# raise UnificationError(f"Cannot unify {expr1} and {expr2}.")

# def unify\_variable(var, expr, substitutions):

# if var in substitutions:

# return unify(substitutions[var], expr, substitutions)

# elif occurs\_check(var, expr, substitutions):

# raise UnificationError(f"Occurs check failed: {var} in {expr}.")

# else:

# substitutions[var] = expr

# return substitutions

# def unify\_lists(list1, list2, substitutions):

# for expr1, expr2 in zip(list1, list2):

# substitutions = unify(expr1, expr2, substitutions)

# return substitutions

# def is\_variable(term):

# return isinstance(term, str) and term[0].islower()

# def is\_compound(term):

# return isinstance(term, (list, tuple)) and len(term) > 0

# def occurs\_check(var, expr, substitutions):

# if var == expr:

# return True

# elif is\_compound(expr):

# return any(occurs\_check(var, sub, substitutions) for sub in expr)

# elif expr in substitutions:

# return occurs\_check(var, substitutions[expr], substitutions)

# return False

# # Example usage

# try:

# expr1 = ("f", "x", ("g", "y"))

# expr2 = ("f", "a", ("g", "b"))

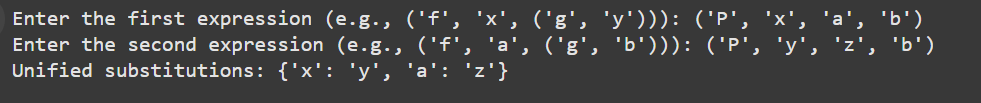
# result = unify(expr1, expr2)

# print("Unified substitutions:", result)

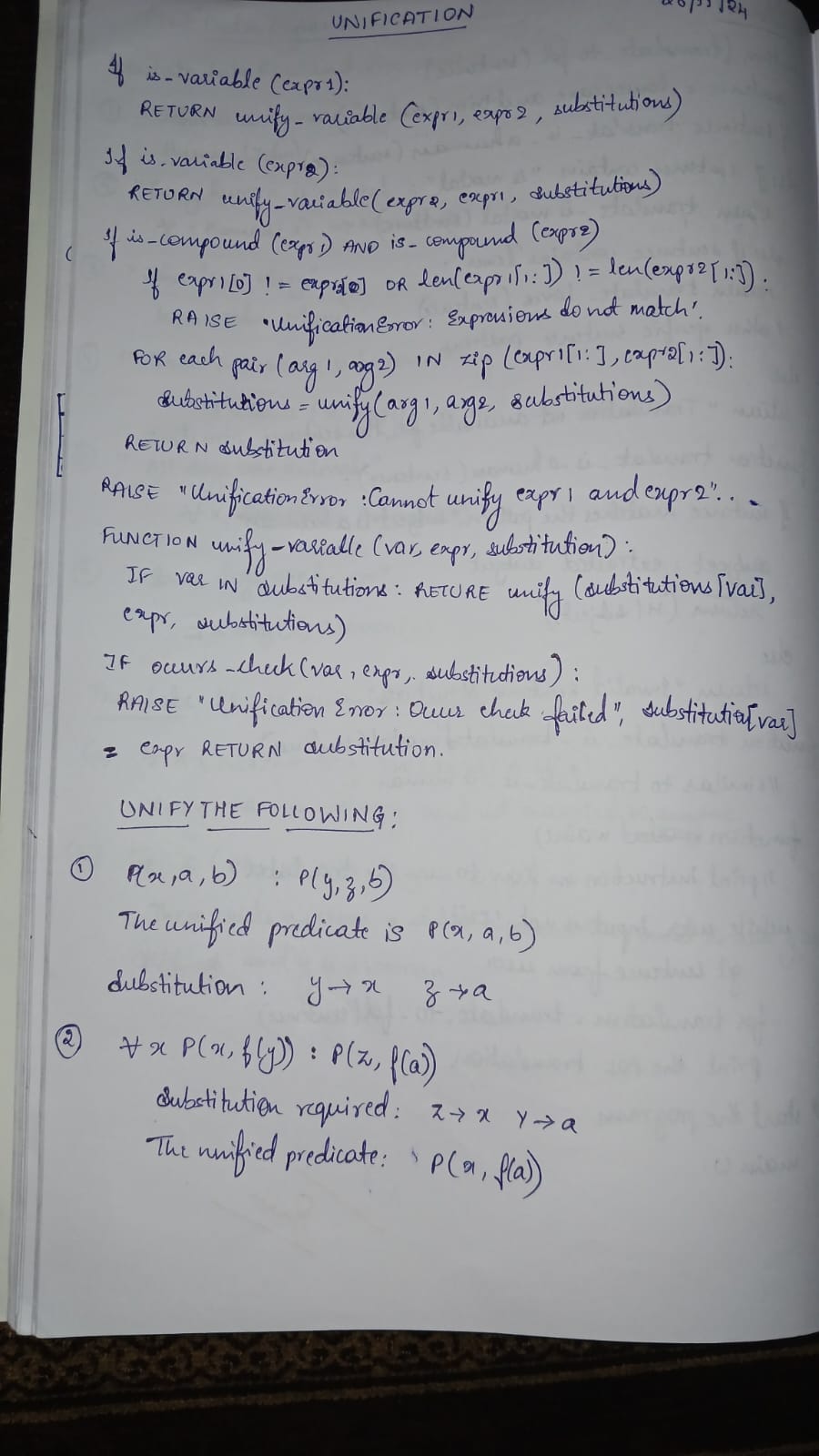
# except UnificationError as e:

# print("Unification failed:", e)

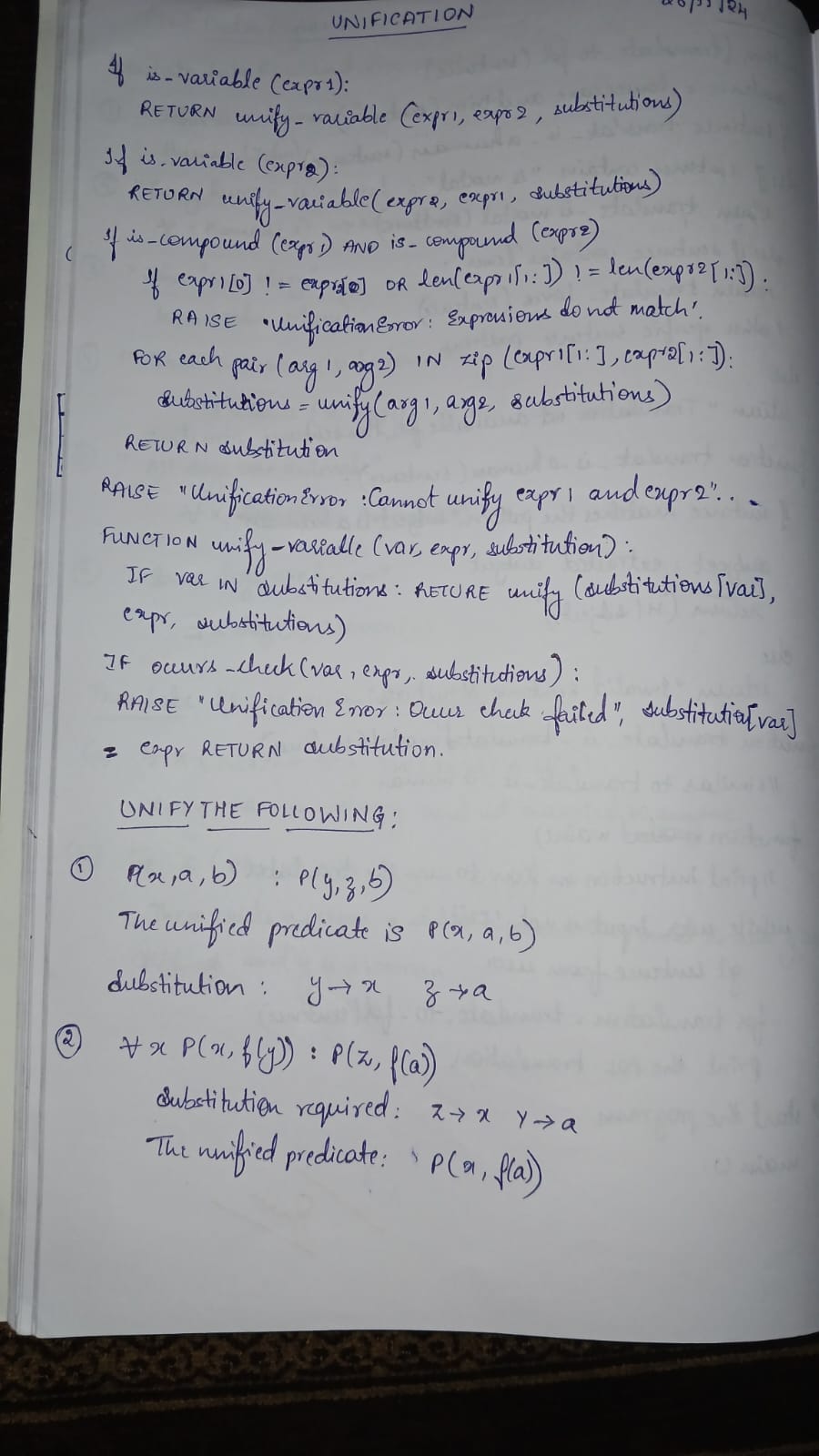
**OUTPUT:**

****

**ALGORITHM:**



**STATE SPACE:**



# FORWARD CHAINING DATE:3-12-24

# facts = {"InAmerica(West)", "SoldWeapons(West, Nono)", "Enemy(Nono,

# America)"}

# rules = [

# {

# "conditions": ["InAmerica(x)", "SoldWeapons(x, y)", "Enemy(y, America)"],

# "conclusion": "Criminal(x)",

# },

# {

# "conditions": ["Enemy(y, America)"],

# "conclusion": "Dangerous(y)",

# },

# ]

# # Forward chaining function

# def forward\_chaining(facts, rules):

# derived\_facts = set(facts)  # Initialize derived facts

# while True:

# new\_fact\_found = False

# for rule in rules:

# # Substitute variables and check if conditions are met

# for fact in derived\_facts:

# if "x" in rule["conditions"][0]:

# # Substitute variables (x, y) with specific instances

# for condition in rule["conditions"]:

# if "x" in condition or "y" in condition:

# x = "West"  # Hardcoded substitution for simplicity

# y = "Nono"

# conditions = [

# cond.replace("x", x).replace("y", y)

# for cond in rule["conditions"]

# ]

# conclusion = (

# rule["conclusion"].replace("x", x).replace("y", y)

# )

# # Check if all conditions are satisfied

# if all(cond in derived\_facts for cond in conditions) and conclusion not in derived\_facts:

# derived\_facts.add(conclusion)

# print(f"New fact derived: {conclusion}")

# new\_fact\_found = True

# 

# # Exit loop if no new fact is found

# if not new\_fact\_found:

# break

# return derived\_facts

# # Run forward chaining

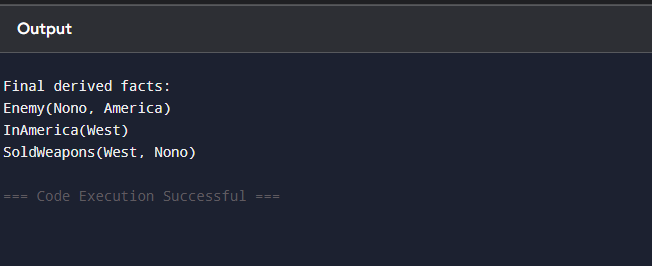
# final\_facts = forward\_chaining(facts, rules)

# print("\nFinal derived facts:")

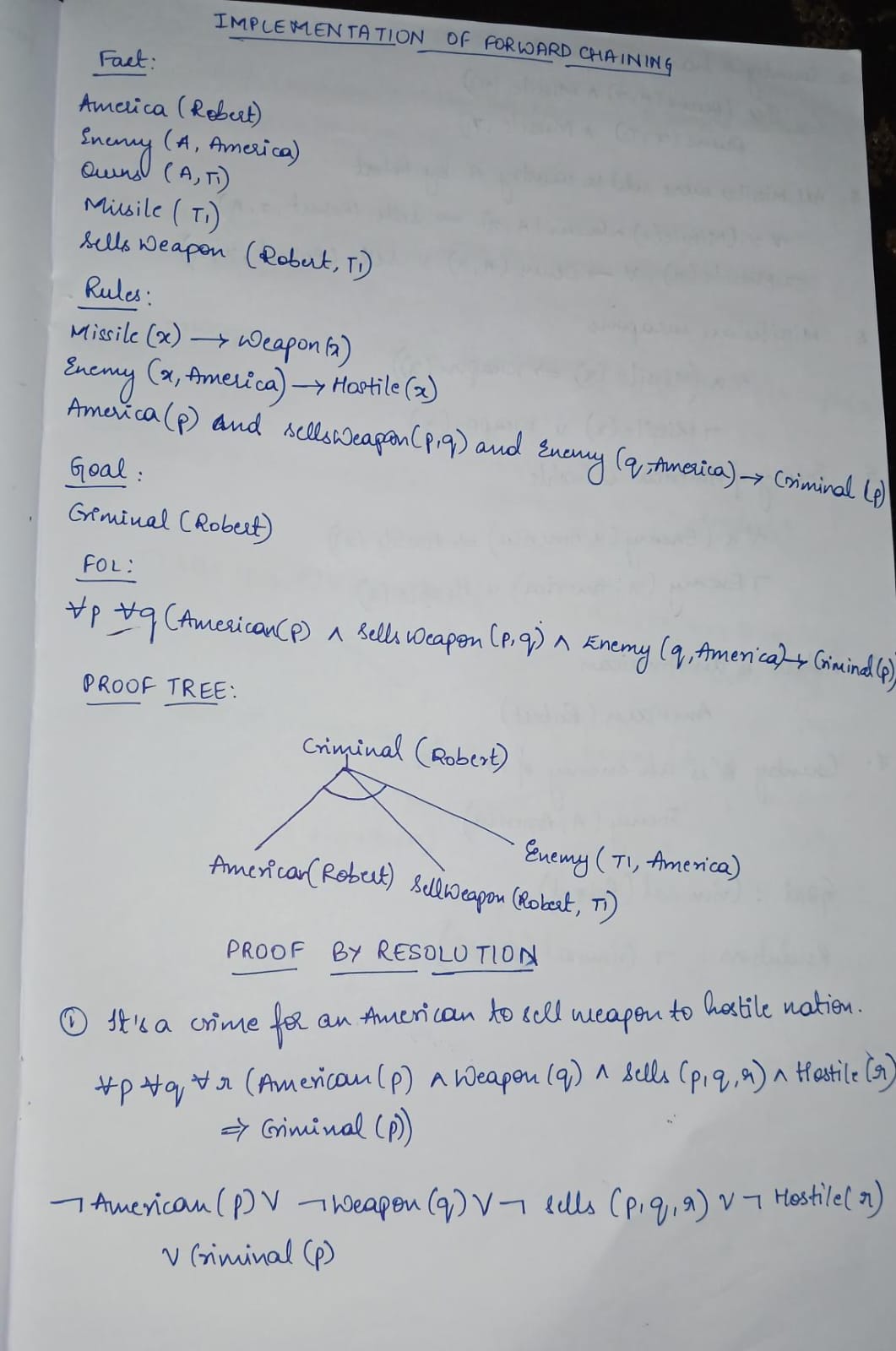
# for fact in final\_facts:

# print(fact)

**OUTPUT:**



**ALGORITHM AND STATE SPACE:**



**RESOLUTION: DATE:3-12-24**

def negate(literal):

    """Return the negation of a literal."""

    if isinstance(literal, tuple) and literal[0] == "not":

        return literal[1]

    else:

        return ("not", literal)

def resolve(clause1, clause2):

    """Return the resolvent of two clauses."""

    resolvents = set()

    for literal1 in clause1:

        for literal2 in clause2:

            if literal1 == negate(literal2):

                resolvent = (clause1 - {literal1}) | (clause2 - {literal2})

                print(f"    Resolving literal: {literal1} with {literal2}")

                print(f"    Resulting Resolvent: {resolvent}")

                resolvents.add(frozenset(resolvent))

    return resolvents

def resolution\_algorithm(KB, query):

    """Perform the resolution algorithm to check if the query can be proven."""

    print("\n--- Step-by-Step Resolution Process ---")

    negated\_query = negate(query)

    KB.append(frozenset([negated\_query]))

    print(f"Negated Query Added to KB: {negated\_query}")

    clauses = set(KB)

    step = 1

    while True:

        new\_clauses = set()

        print(f"\nStep {step}: Resolving Clauses")

        for c1 in clauses:

            for c2 in clauses:

                if c1 != c2:

                    print(f"  Resolving clauses: {c1} and {c2}")

                    resolvent = resolve(c1, c2)

                    for res in resolvent:

                        if frozenset([]) in resolvent:

                            print("\nEmpty clause derived! The query is provable.")

                            return True

                        new\_clauses.add(res)

        if new\_clauses.issubset(clauses):

            print("\nNo new clauses can be derived. The query is not provable.")

            return False

        clauses.update(new\_clauses)

        step += 1

KB = [

    frozenset([("not", "food(x)"), ("likes", "John", "x")]),

    frozenset([("food", "Apple")]),

    frozenset([("food", "vegetables")]),

    frozenset([("not", "eats(y, z)"), ("killed", "y"), ("food", "z")]),

    frozenset([("eats", "Anil", "Peanuts")]),

    frozenset([("alive", "Anil")]),

    frozenset([("not", "eats(Anil, w)"), ("eats", "Harry", "w")]),

    frozenset([("killed", "g"), ("alive", "g")]),

    frozenset([("not", "alive(k)"), ("not", "killed(k)")]),

    frozenset([("likes", "John", "Peanuts")])

]

query = ("likes", "John", "Peanuts")

result = resolution\_algorithm(KB, query)

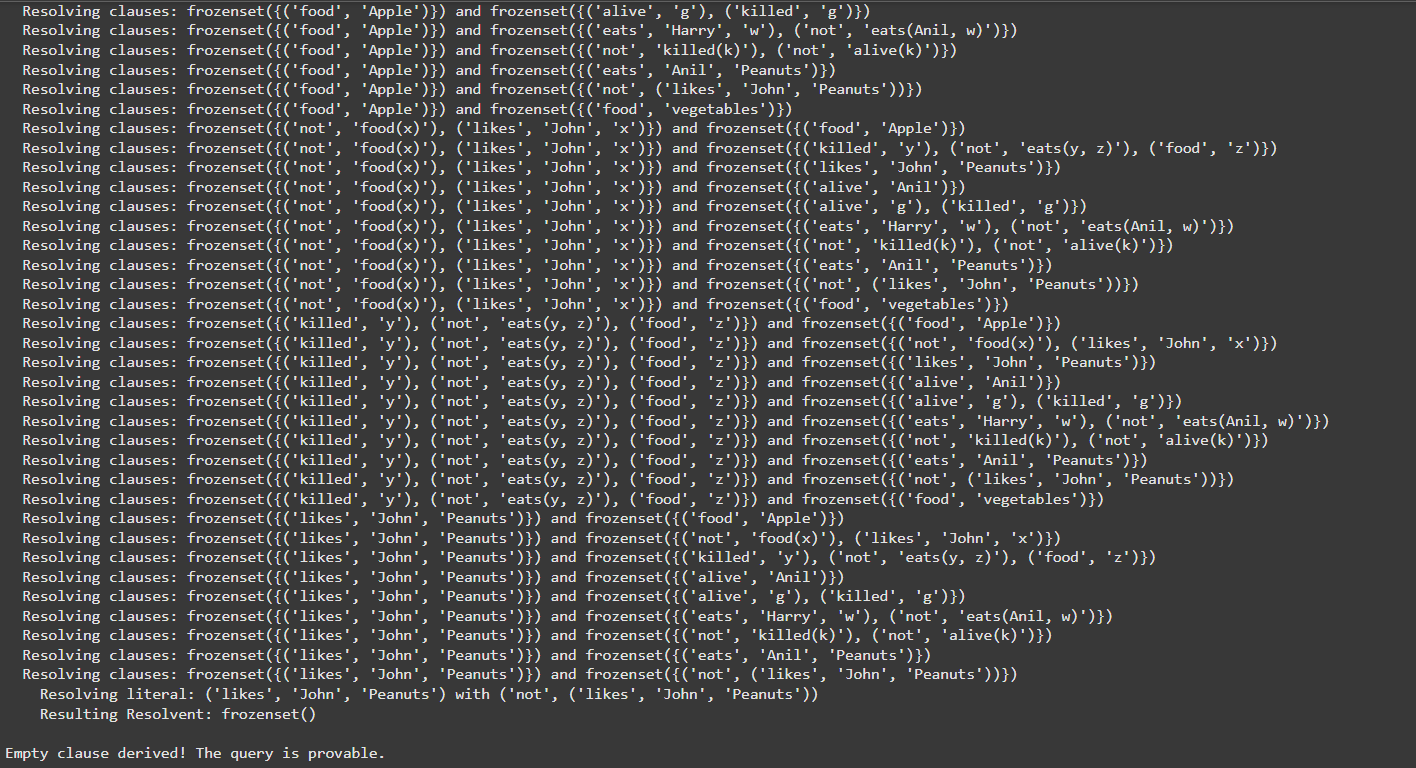
if result:

    print("\nQuery is provable.")

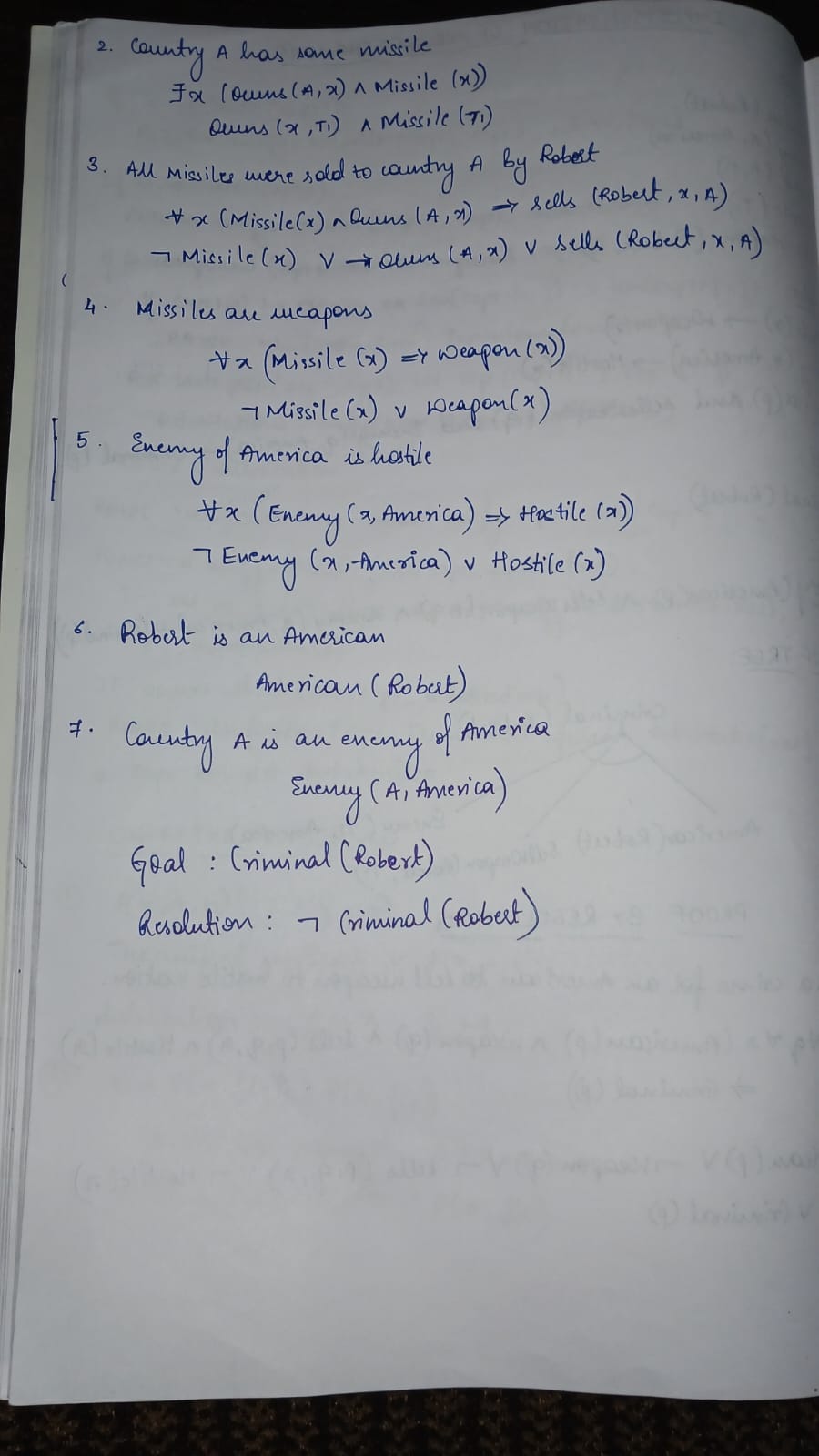
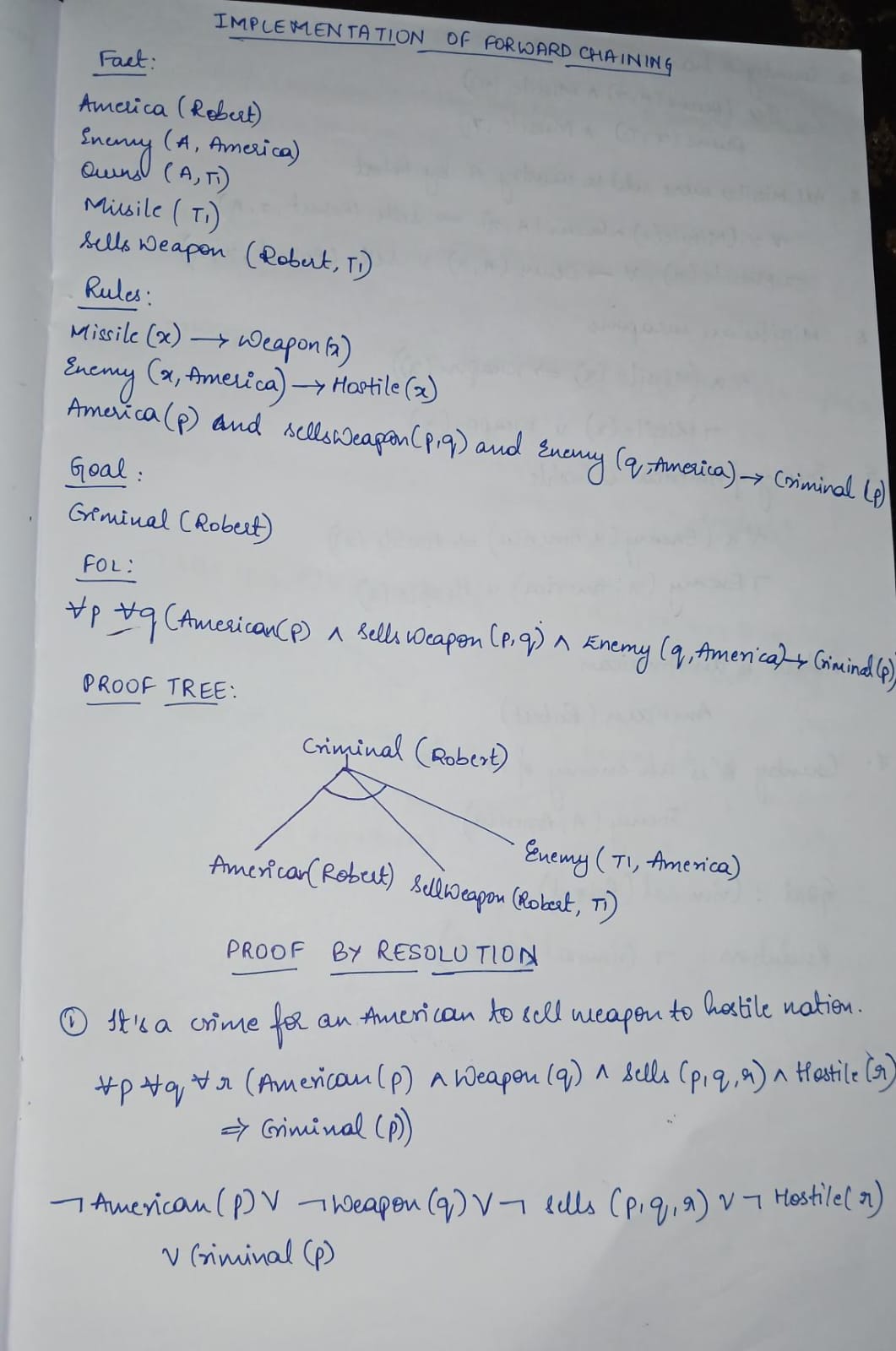
else:

    print("\nQuery is not provable.")

**OUTPUT:**



**ALGORITHM AND STATE SPACE:**



**ALPHA BETA PRUNING DATE:17-12-24**

def alpha\_beta\_search(state):

    def max\_value(state, alpha, beta, path):

        print(f"MAX: Visiting state {state}, alpha={alpha}, beta={beta}")

        if terminal\_test(state):

            print(f"MAX: Terminal state {state} has utility {utility(state)}")

            return utility(state), path

        v = float('-inf')

        best\_path = []

        for action in actions(state):

            result\_state = result(state, action)

            value, new\_path = min\_value(result\_state, alpha, beta, path + [action])

            print(f"MAX: From state {state}, action {action} → value={value}")

            if value > v:

                v = value

                best\_path = new\_path

            if v >= beta:

                print(f"MAX: Pruning at state {state} with value={v} ≥ beta={beta}")

                return v, best\_path

            alpha = max(alpha, v)

        print(f"MAX: Returning value={v} for state {state}")

        return v, best\_path

    def min\_value(state, alpha, beta, path):

        print(f"MIN: Visiting state {state}, alpha={alpha}, beta={beta}")

        if terminal\_test(state):

            print(f"MIN: Terminal state {state} has utility {utility(state)}")

            return utility(state), path

        v = float('inf')

        best\_path = []

        for action in actions(state):

            result\_state = result(state, action)

            value, new\_path = max\_value(result\_state, alpha, beta, path + [action])

            print(f"MIN: From state {state}, action {action} → value={value}")

            if value < v:

                v = value

                best\_path = new\_path

            if v <= alpha:

                print(f"MIN: Pruning at state {state} with value={v} ≤ alpha={alpha}")

                return v, best\_path

            beta = min(beta, v)

        print(f"MIN: Returning value={v} for state {state}")

        return v, best\_path

    print("Starting Alpha-Beta Search...\n")

    final\_value, final\_path = max\_value(state, float('-inf'), float('inf'), [state])

    return final\_value, final\_path

def terminal\_test(state):

    return state in terminal\_states

def utility(state):

    return terminal\_states[state]

def actions(state):

    return game\_tree.get(state, [])

def result(state, action):

    return action

game\_tree = {

    'A': ['B', 'C'],

    'B': ['D', 'E'],

    'C': ['F', 'G'],

    'D': ['H','I'],

    'E': ['J','K'],

    'F': ['L','M'],

    'G': ['N','O'],

    'H':[],

    'I':[],

    'J':[],

    'K':[],

    'L':[],

    'M':[],

    'N':[],

    'O':[]

}

terminal\_states = {

    'H': 10,

    'I': 9,

    'J': 14,

    'K': 18,

    'L': 5,

    'M': 4,

    'N': 50,

    'O': 3

}

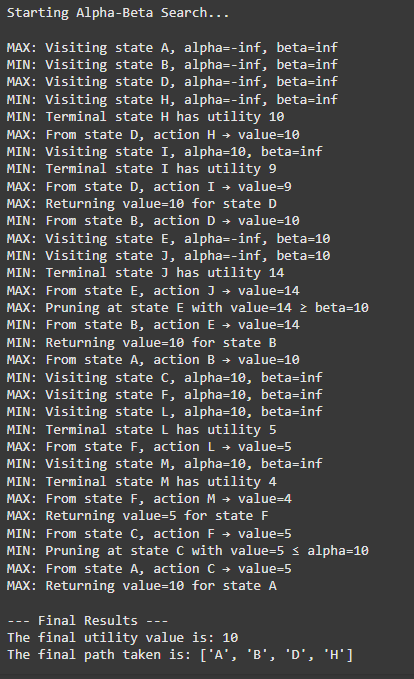
final\_value, final\_path = alpha\_beta\_search('A')

print("\n--- Final Results ---")

print(f"The final utility value is: {final\_value}")

print(f"The final path taken is: {final\_path}")

**OUTPUT:**



**ALGORITHM AND STATE SPACE:**

