VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shraddha (1BM22CS357)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

Prof. Swathi Sridharan
Professor
Department of Computer Science and Engineering

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried outby Shraddha **(1BM22CS357),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sneha P Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

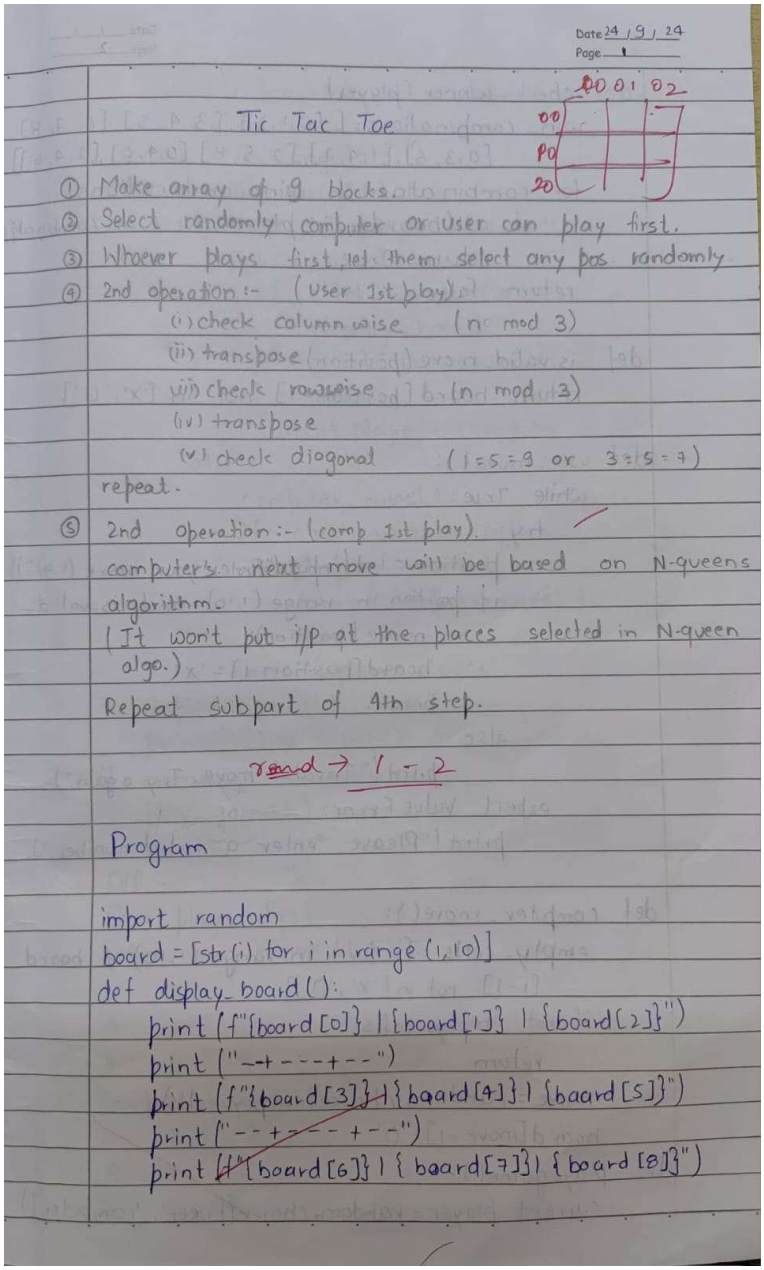
Index

Github Link: https://github.com/Shraddha-357/AI_lab

Program1

Tic-Tac-Toe:

Algorithm:



Date / /
Page 2

```
def check_winner(player):
    win_combinations = [[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 combination in win_combinations:
        if all(board[i]==player for i in combination):
            return True
    return False.

def is_valid_move(position):
    return board[position-1] not in ['X','O']

def user_move():
    while True:
        try:
            position = int(input("Enter your move (1-9)"))
            if position in range(1,10) and is_valid_
                move(position):
                board[position-1] = 'X'
                break.
            else:
                print("Invalid move. Try again.")
        expect ValueError:
            print("Please enter a valid number.").

def computer_move():
    empty_cells = [i for i in range(1,10) if board
        [i-1] not in ['X','O']]
    if not empty_cells:
        return
    move = random.choice(empty_cells)
    board[move-1] = 'O'.
def play_game():
    current_player = random.choice['user','computer']
```

Date __/__/__
Page 3

```
printf(f"{current_player.capitalise()} plays first!")

for _ in range(9):
    display_board()
    if current_player == 'user':
        user_move()
        if check_winner('X'):
            display_board()
            print("User wins!")
            return
        current_player = "computer".
    else:
        computer_move()
        if check_winner('O'):
            display_board()
            print("Computer wins!")
            return
        current_player = 'user'.
display_board()
print("It's a tie!")
if __name__ == "__main__":
    play_game().
```

O/P :-

User plays first.

Enter your move (1-9): 5

1 2 3

4 X 6

7 8 9

Computer move: 8

1 2 3

4 X 6

7 O 9.

Date / /
Page 4

Enter your move: 1

X 2 3
4 X 6
7 0 9

Computer move: 9

X 2 3
4 X 6
7 0 0

Enter your move: 7

X 2 3
4 X 6
X 0 0

Computer move: 4

X 2 3
0 X 6
X 0 0

Enter your more: 3

X 2 X
0 X 6
X 0 0

Player wins!!

29/9/24

Code:

```
import random

# Initialize the board
board = [[" for _ in range(3)] for _ in range(3)]

# Function to print the board
def print_board():
    for row in board:
        print('|'.join(cell if cell else ' ' for cell in row))
    print('-' * 9)

# Function to check if a player has won
def check_win(player):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] == player:  # Check rows
            return True
        if board[0][i] == board[1][i] == board[2][i] == player:  # Check columns
            return True
    if board[0][0] == board[1][1] == board[2][2] == player:  # Check diagonal
        return True
    if board[0][2] == board[1][1] == board[2][0] == player:  # Check anti-diagonal
        return True
    return False

# Function to check if the game is a draw
def check_draw():
    for row in board:
        if " in row:
            return False
    return True

# Computer's move
def computer():
    # Try to win
    for row in range(3):
        for col in range(3):
            if board[row][col] == ":
                board[row][col] = 'O'
                if check_win('O'):
                    return
                board[row][col] = "
    # Block opponent
    for row in range(3):
        for col in range(3):
            if board[row][col] == ":
```

```
            board[row][col] = 'X'
            if check_win('X'):
                board[row][col] = 'O'
                return
            board[row][col] = ''
    # Take center
    if board[1][1] == '':
        board[1][1] = 'O'
        return
    # Take corners
    corners = [(0, 0), (0, 2), (2, 0), (2, 2)]
    random.shuffle(corners)
    for row, col in corners:
        if board[row][col] == '':
            board[row][col] = 'O'
            return
    # Take sides
    sides = [(0, 1), (1, 0), (1, 2), (2, 1)]
    random.shuffle(sides)
    for row, col in sides:
        if board[row][col] == '':
            board[row][col] = 'O'
            return

# Main game loop
flag = True
while flag:
    print_board()
    while True:
        try:
            print("Player 1's turn (X)")
            user_input = input("Enter row and column (1-3) separated by space: ")
            row, col = map(int, user_input.split())
            row -= 1
            col -= 1
            if row in range(3) and col in range(3) and board[row][col] == '':
                board[row][col] = 'X'
                break
            else:
                print("Invalid move, cell already taken or out of bounds.")
        except (ValueError, IndexError):
            print("Invalid input, please enter two numbers between 1 and 3 separated by space.")

    if check_win('X'):
        print_board()
        print("Player 1 wins!")
        break
    if check_draw():
```

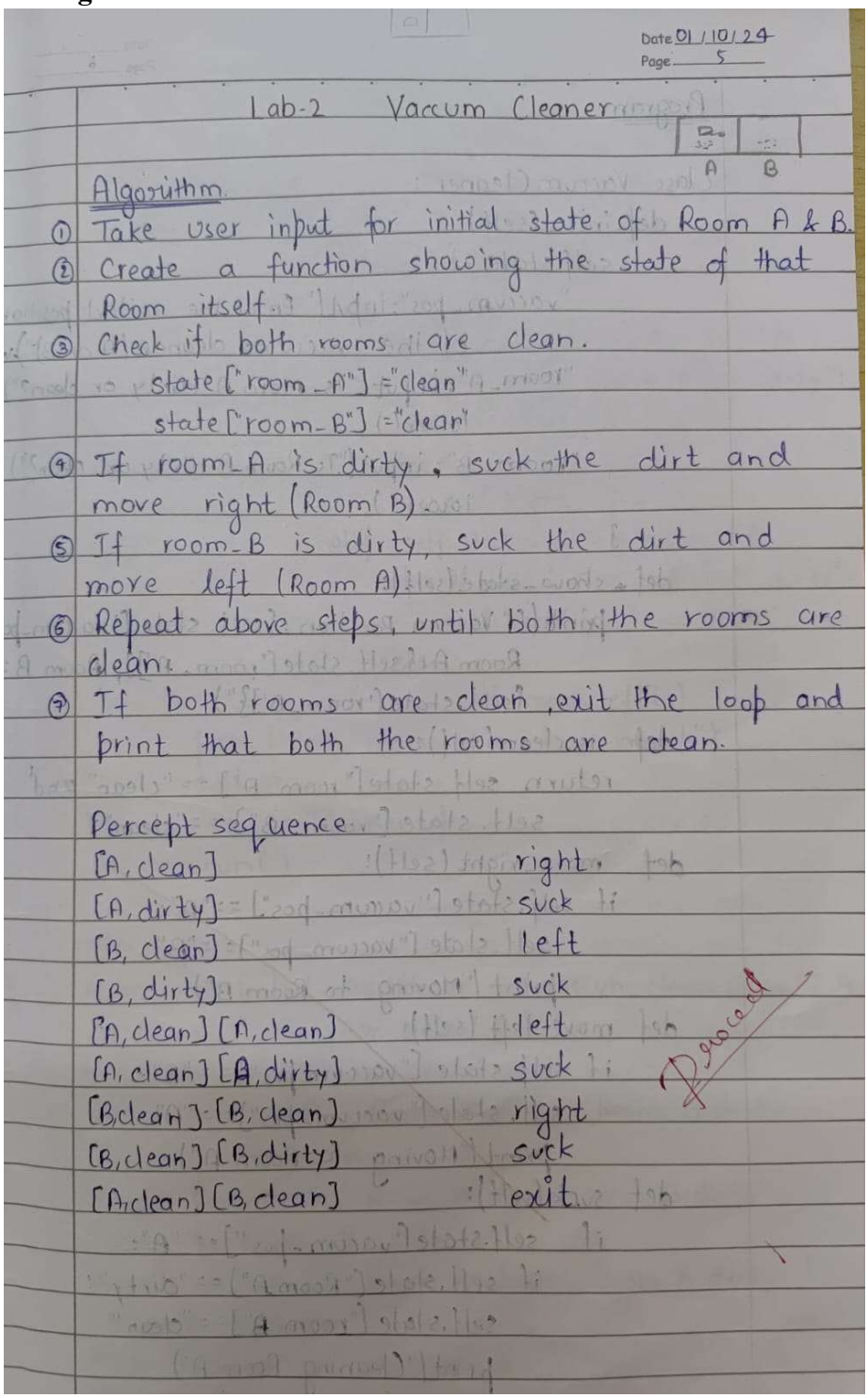
```
    print_board()
    print("Draw!")
    break

print_board()
print("Player 2's turn (O)")
computer()
if check_win('O'):
    print_board()
    print("Player 2 wins!")
    break
if check_draw():
    print_board()
    print("Draw!")
    break
```

Output:

```
Player 1's turn (X)
Enter row and column (1-3) separated by space: 1 2
X | X | O
---------
O | X | X
---------
  |   | O
---------
Player 2's turn (O)
X | X | O
---------
O | X | X
---------
  | O | O
---------
Player 1's turn (X)
Enter row and column (1-3) separated by space: 3 1
X | X | O
---------
O | X | X
---------
X | O | O
---------
Draw!
```

VacuumCleanerAgent:

Algorithm:



Date 01/10/24
Page 5

Lab-2 Vaccum Cleaner

A B

Algorithm

① Take user input for initial state of Room A & B.
② Create a function showing the state of that Room itself.
③ Check if both rooms are clean.
state["room_A"] = "clean"
state["room_B"] = "clean"
④ If room_A is dirty, suck the dirt and move right (Room B).
⑤ If room_B is dirty, suck the dirt and move left (Room A).
⑥ Repeat above steps, until both the rooms are clean.
⑦ If both rooms are clean, exit the loop and print that both the rooms are clean.

Percept sequence

[A, clean]	right
[A, dirty]	suck
[B, clean]	left
[B, dirty]	suck
[A, clean] [A, clean]	left
[A, clean] [A, dirty]	suck
[B, clean] [B, clean]	right
[B, clean] [B, dirty]	suck
[A, clean] [B, clean]	exit

Proceed

Date / /
Page 6

Program

```
class VaccumCleaner:
    def init_(self):
        self.state = {
            "vaccum_pos": input("Enter the initial position
                of the vaccum cleaner (A or B):").upper(),
            "room_A": input("Is room A dirty or clean?")
                .lower(),
            "room_B": input("Is Room B dirty or clean?").
                lower(),
        }
    def show_state(self):
        print(f"Vaccum Position: {self.state['vaccum_pos']},
            Room A: {self.state['room_A']}, Room B:
            {self.state['room_B']}")
    def is_clean(self):
        return self.state["room_A"] == "clean" and
            self.state["room_B"] == "clean"
    def move_right(self):
        if self.state["vaccum_pos"] == "A":
            self.state["vaccum_pos"] = "B"
            print("Moving to Room B").
    def move_left(self):
        if self.state["vaccum_pos"] == "B":
            self.state["vaccum_pos"] = "A":
            print("Moving to Room A").
    def suck(self):
        if self.state["vacum_pos"] == "A":
            if self.state["Room_A"] == "dirty":
                self.state["room_A"] == "clean"
                print("Cleaning Room A").
        else self.state["vacum_pos"] == "B":
```

```
            if self.state["room_B"] == "dirty":
                self.state["room_B"] = "clean"
                print("Cleaning Room B")
    def run(self):
        while not self.is_clean():
            self.show_state()
            if self.state["vaccum_pos"] == "A":
                if self.state["Room_A"] == "dirty":
                    self.suck()
                else:
                    self.move_right()
            elif self.state["vaccum_pos"] == "B":
                if self.state["room_B"] == "dirty":
                    self.suck()
                else:
                    self.move_left()
        print("Both rooms are clean now!")
        self.show_state()
vaccum = VaccumCleaner()
vaccum.run()
```

O/P:-

```
Enter the initial position of the vaccum cleane (A or B): A
Is Room A dirty or clean? dirty
Is Room B dirty or clean? dirty
Vaccum Position: A, Room A: dirty, Room B: dirty
Cleaning Room A
Vaccum Position: A, Room A: clean, Room B: dirty
Moving to Room B
Vaccum Position: B, Room A: clean, Room B: dirty
Cleaning Room B
Both rooms are clean now!
Vaccum Position: B, Room A: clean, Room B: clean
```

For 4 rooms:-

Changes in code.

```
self.neighbours = {
        'A': {'right': 'B', 'down': 'C'}
        'B': {'left': 'A', 'down': 'D'}
        'C': {'right': 'D', 'up': 'A'}
        'D': {'left': 'C', 'up': 'B'}.

def move(self, direction):
    if direction in self.neighbors[self.location]:
        self.location: self.neighbors[self.location][direction]
        print(f"moved {direction} to room {self.location}")
    else:
        print(f"Cannot move{direction} from room
            {self.location}")
```

O/P:-

Vaccum is in room A, Room state: clean
Action: Move down
Move down to Room C
Room state: {'A': 'clean', 'B', : 'clean', 'C': dirty', 'D': dirty'}.
Room state: {'A': 'clean', 'B', : 'clean', C: clean, 'D': 'dirty'}.
Room state: {'A': clean, B: clean, C: clean, D: clean}
All rooms are clean.

Code:

```
class VacuumCleaner:
    def _init_(self):
        self.state = {
            "vacuum_pos": input("Enter the initial position of the vacuum cleaner (A or B): ").upper(),
            "room_A": input("Is Room A dirty or clean? ").lower(),
            "room_B": input("Is Room B dirty or clean? ").lower()
        }

    def show_state(self):
        print(f"Vacuum Position: {self.state['vacuum_pos']}, Room A: {self.state['room_A']}, Room B:
{self.state['room_B']}")

    def is_clean(self):
        return self.state["room_A"] == "clean" and self.state["room_B"] == "clean"

    def move_right(self):
        if self.state["vacuum_pos"] == "A":
            self.state["vacuum_pos"] = "B"
            print("Moving to Room B")

    def move_left(self):
        if self.state["vacuum_pos"] == "B":
            self.state["vacuum_pos"] = "A"
            print("Moving to Room A")

    def suck(self):
        if self.state["vacuum_pos"] == "A":
            if self.state["room_A"] == "dirty":
                self.state["room_A"] = "clean"
                print("Cleaning Room A")
        elif self.state["vacuum_pos"] == "B":
            if self.state["room_B"] == "dirty":
                self.state["room_B"] = "clean"
                print("Cleaning Room B")

    def run(self):
        while not self.is_clean():
            self.show_state()
            if self.state["vacuum_pos"] == "A":
                if self.state["room_A"] == "dirty":
                    self.suck()
                else:
                    self.move_right()
            elif self.state["vacuum_pos"] == "B":
                if self.state["room_B"] == "dirty":
```

```
                self.suck()
            else:
                self.move_left()
        print("Both rooms are clean now!")
        self.show_state()


# Create Vacuum Cleaner object and run the simulation
vacuum = VacuumCleaner()
vacuum.run
```

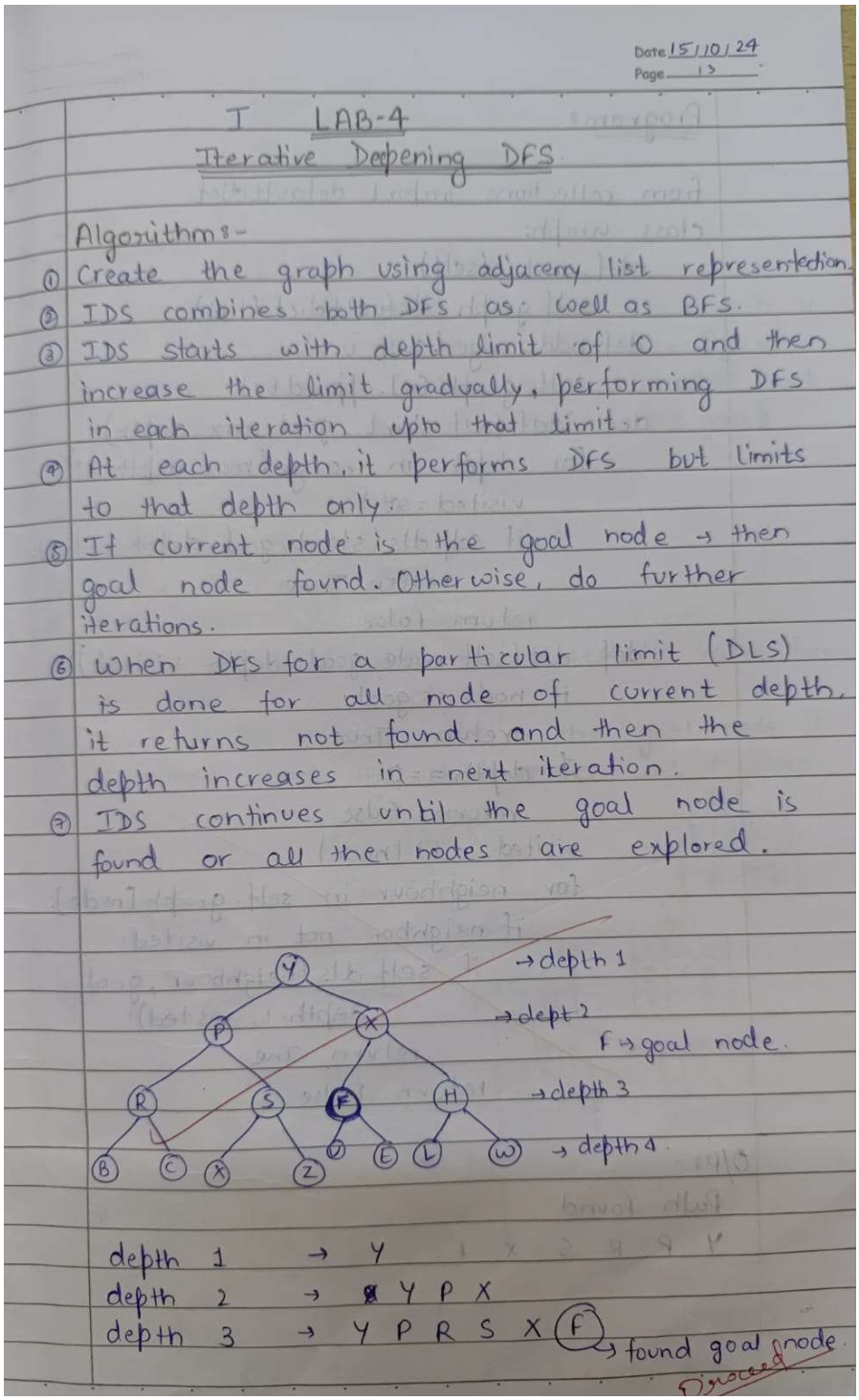
Output:

```
Enter the initial position of the vacuum cleaner (A or B): A
Is Room A dirty or clean? dirty
Is Room B dirty or clean? dirty
Vacuum Position: A, Room A: dirty, Room B: dirty
Cleaning Room A
Vacuum Position: A, Room A: clean, Room B: dirty
Moving to Room B
Vacuum Position: B, Room A: clean, Room B: dirty
Cleaning Room B
Both rooms are clean now!
Vacuum Position: B, Room A: clean, Room B: clean

=== Code Execution Successful ===
```

Program2

Iterative deepening search depth first search:

Algorithm:



Date 15/10/24
Page 13

I LAB-4
Iterative Deepening DFS

Algorithm:-
① Create the graph using adjacency list representation.
② IDS combines both DFS as well as BFS.
③ IDS starts with depth limit of 0 and then increase the limit gradually, performing DFS in each iteration upto that limit.
④ At each depth, it performs DFS but limits to that depth only.
⑤ If current node is the goal node → then goal node found. Otherwise, do further iterations.
⑥ When DFS for a particular limit (DLS) is done for all node of current depth, it returns not found. and then the depth increases in next iteration.
⑦ IDS continues until the goal node is found or all the nodes are explored.

depth 1 → Y
depth 2 → Y P X
depth 3 → Y P R S X F → found goal node.

Discard

Program:-

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
    def iddfs(self, start, goal, max_depth):
        for depth in range(max_depth + 1):
            visited = set()
            if self.dls(start, goal, depth, visited):
                return True
            return False
    def dls(self, node, goal, depth, visited):
        if node == goal:
            return True
        if depth == 0:
            return False
        visited.add(node)
        for neighbour in self.graph[node]:
            if neighbor not in visited:
                if self.dls(neighbour, goal,
                            depth-1, visited):
                    return True
        return False
```

O/P:-

Path found.

Y P R S X F.

Code:

```
from collections import defaultdict

# This class represents a directed graph using adjacency list representation
class Graph:

    def _init_(self, vertices):
        # No. of vertices
        self.V = vertices

        # Default dictionary to store graph
        self.graph = defaultdict(list)

    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function to perform a Depth-Limited search from given source 'src'
    def DLS(self, src, target, maxDepth, depth):
        print(f"Depth {depth}: Exploring {src}")

        if src == target:
            print(f"Depth {depth}: Found target {target}")
            return True

        # If reached the maximum depth, stop recursing.
        if maxDepth <= 0:
            return False

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[src]:
            if self.DLS(i, target, maxDepth - 1, depth + 1):
                return True
        return False

    # IDDFS to search if target is reachable from src. It uses recursive DLS()
    def IDDFS(self, src, target, maxDepth):
        # Repeatedly depth-limit search till the maximum depth
        for i in range(maxDepth + 1):
            print(f"Starting iteration with depth limit: {i}")
            if self.DLS(src, target, i, 0):
                return True
            print(f"Iteration with depth limit {i} did not find the target.\n")
        return False

# Create a graph given in the above diagram
```

```
g = Graph(7)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

target = 6
maxDepth = 3
src = 0

if g.IDDFS(src, target, maxDepth) == True:
    print("Target is reachable from source within max depth")
else:
    print("Target is NOT reachable from source within max depth")
```

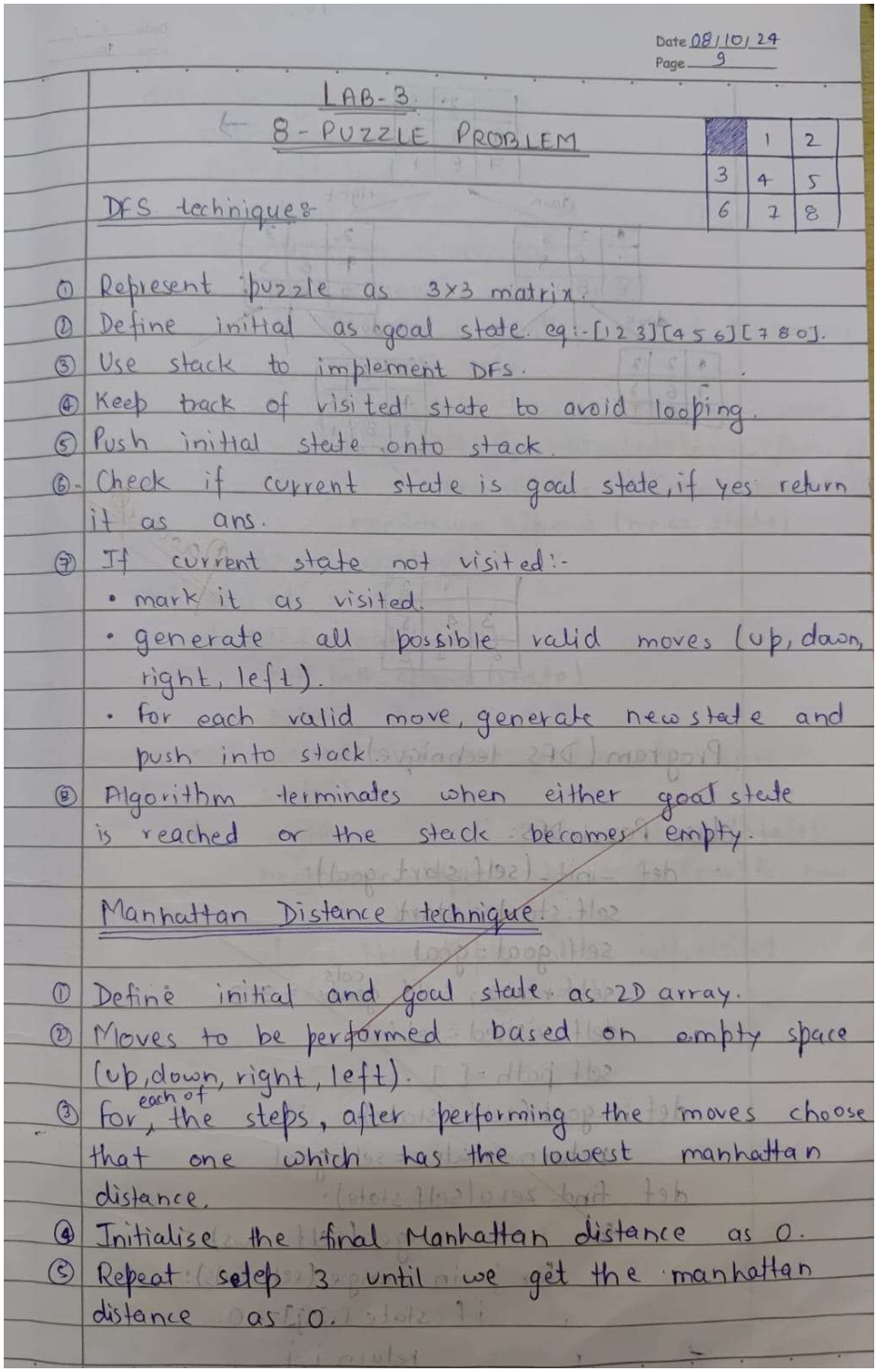
Output:

```
Starting iteration with depth limit: 0
Depth 0: Exploring 0
Iteration with depth limit 0 did not find the target.

Starting iteration with depth limit: 1
Depth 0: Exploring 0
Depth 1: Exploring 1
Depth 1: Exploring 2
Iteration with depth limit 1 did not find the target.

Starting iteration with depth limit: 2
Depth 0: Exploring 0
Depth 1: Exploring 1
Depth 2: Exploring 3
Depth 2: Exploring 4
Depth 1: Exploring 2
Depth 2: Exploring 5
Depth 2: Exploring 6
Depth 2: Found target 6
Target is reachable from source within max depth

=== Code Execution Successful ===
```

Program3

8 puzzle using ManhattanDistance:

Algorithm:



Date 08/10/24
Page 9

LAB-3

8-PUZZLE PROBLEM

	1	2
3	4	5
6	7	8

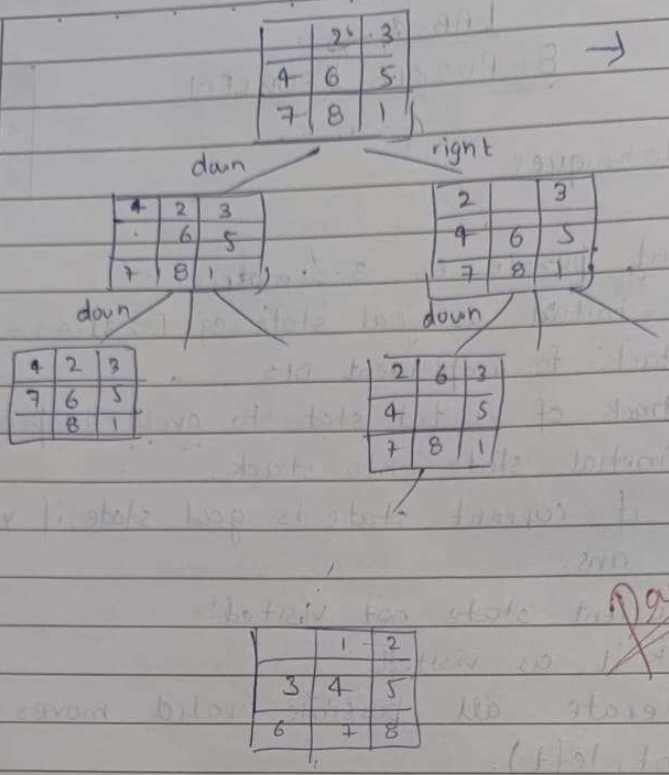
DFS technique:

① Represent puzzle as 3×3 matrix.
② Define initial as goal state. eg:-[1 2 3][4 5 6][7 8 0].
③ Use stack to implement DFS.
④ Keep track of visited state to avoid looping.
⑤ Push initial state onto stack.
⑥ Check if current state is goal state, if yes return it as ans.
⑦ If current state not visited:-
- mark it as visited.
- generate all possible valid moves (up, down, right, left).
- for each valid move, generate new state and push into stack.

⑧ Algorithm terminates when either goal state is reached or the stack becomes empty.

Manhattan Distance technique:

① Define initial and goal state as 2D array.
② Moves to be performed based on empty space (up, down, right, left).
③ For each of the steps, after performing the moves choose that one which has the lowest manhattan distance.
④ Initialise the final Manhattan distance as 0.
⑤ Repeat setep 3 until we get the manhattan distance as 0.



Program (DFS technique):-

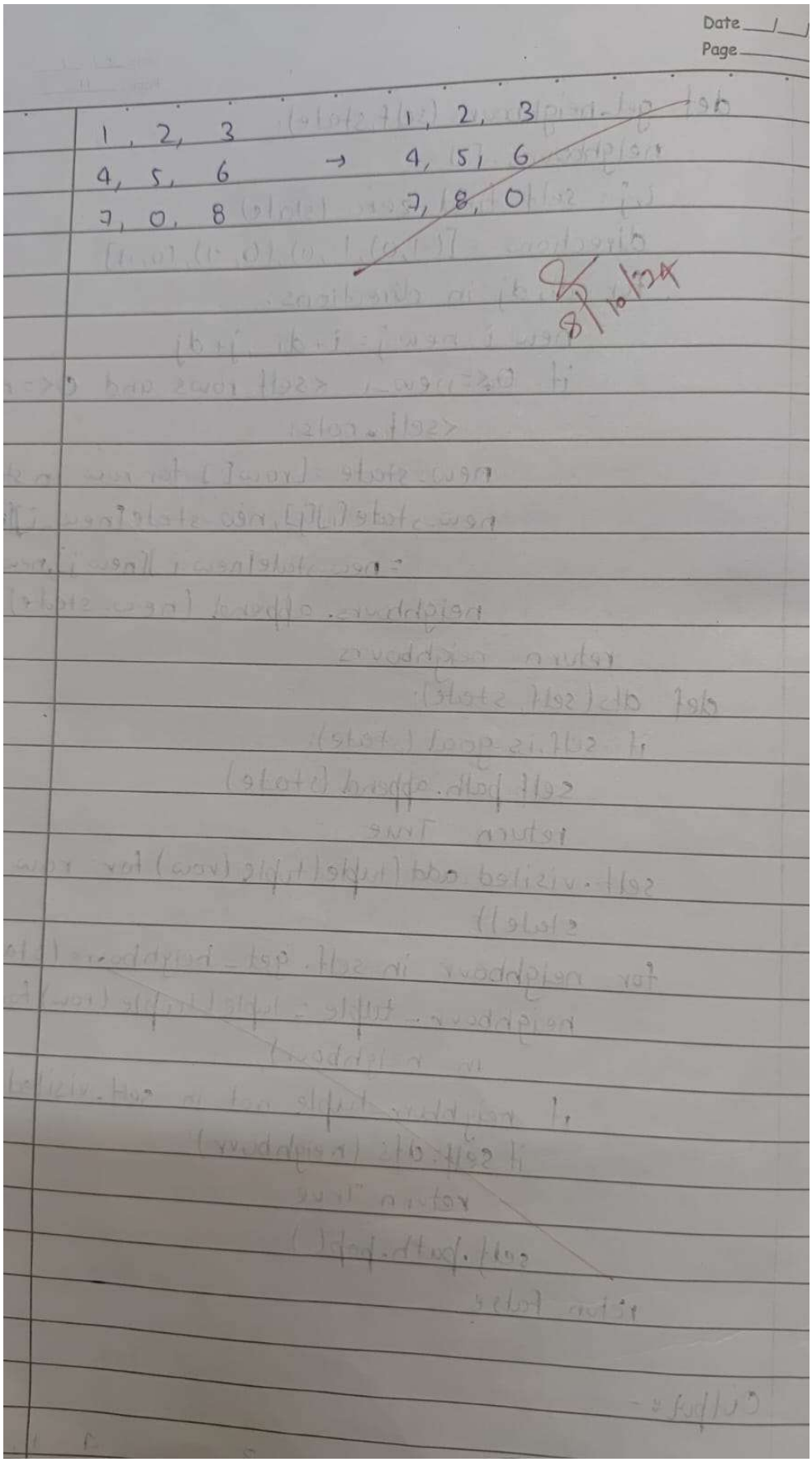
```
class PuzzleDFS:
    def __init__(self, start, goal):
        self.start = start
        self.goal = goal
        self.rows, self.cols = 3, 3
        self.visited = set()
        self.path = []
    def is_goal(self, state):
        return state == self.goal
    def find_zero(self, state):
        for i in range(self.rows):
            for j in range(self.cols):
                if state[i][j] == 0:
                    return i, j
```

Date __/__/__
Page 11

```
def get_neighbours(self,state):
    neighbours=[]
    i,j= self.find_zero(state)
    directions =[(-1,0),(1,0),(0,-1),(0,1)]
    for di,dj in directions:
        new_i, new_j= i+di ,j+dj
        if 0<=new_i <self.rows and 0<=new_j
            <self.cols:
            new_state =[row[:] for row in state]
            new_state[i][j], new_state[new_i][new_j]
                = new_state[new_i][new_j],new_state[i][j]
            neighbours.append(new_state)
    return neighbours.
def dfs(self, state):
    if self.is_goal(state):
        self.path.append(state)
        return True.
    self.visited.add(tuple(tuple(row) for row in
        state))
    for neighbour in self.get_neighbours(state):
        neighbour_tuple = tuple(tuple(row) for row
            in neighbour)
        if neighbour_tuple not in self.visited:
            if self.dfs(neighbour):
                return True
            self.path.pop()
    return false
```

Output:-

```
4, 1, 3          4, 1, 3          4, 1, 3
7, 2, 6    →     7, 2, 6    →     7, 2, 6
5, 8, 0          5, 0, 8          0, 5, 8
```


Date
Page
1, 2, 3
4, 5, 6
7, 0, 8
1, 2, 3
4, 5, 6
7, 8, 0

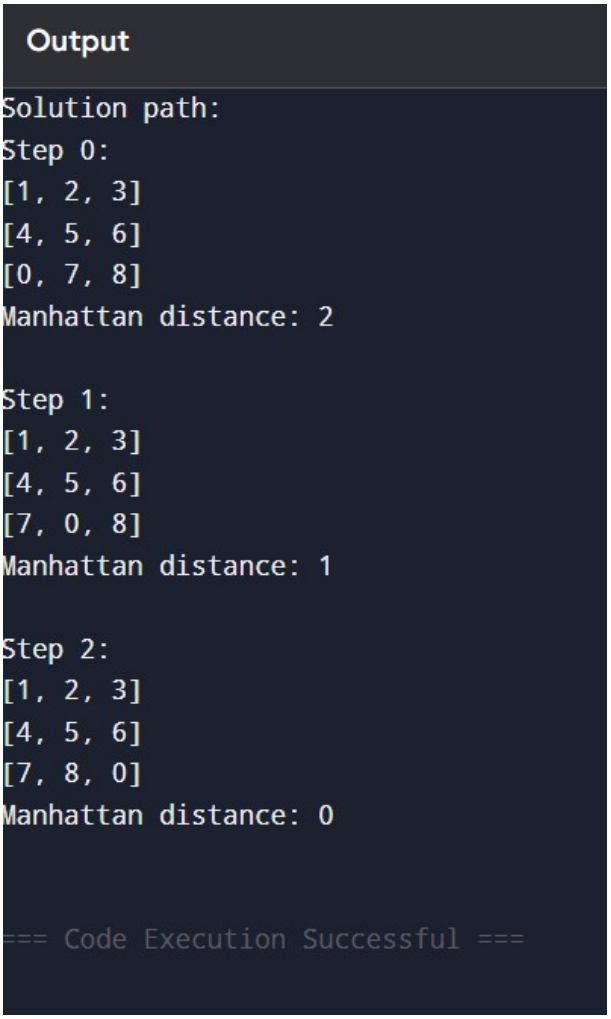
Code:

```
from copy import deepcopy

# Function to calculate Manhattan distance
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:  # Ignore the blank tile
                x_goal, y_goal = divmod(goal_state.index(state[i][j]), 3)
                distance += abs(i - x_goal) + abs(j - y_goal)
    return distance

# Function to find neighbors (valid moves)
def generate_neighbors(state):
    neighbors = []
    blank_x, blank_y = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Right, Left, Down, Up

    for dx, dy in moves:
        new_x, new_y = blank_x + dx, blank_y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = deepcopy(state)
            new_state[blank_x][blank_y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[blank_x][blank_y]
            neighbors.append(new_state)
    return neighbors

# Function to solve the 8-puzzle using Manhattan distance
def solve_8_puzzle(initial_state, goal_state):
    current_state = initial_state
    visited = set()
    path = [deepcopy(initial_state)]
    manhattan_distances = [manhattan_distance(current_state, sum(goal_state, []))]

    while current_state != goal_state:
        visited.add(tuple(map(tuple, current_state)))
        neighbors = generate_neighbors(current_state)
```

```
        min_distance = float('inf')
        next_state = None

        for neighbor in neighbors:
            if tuple(map(tuple, neighbor)) not in visited:
                distance = manhattan_distance(neighbor, sum(goal_state, []))
                if distance < min_distance:
                    min_distance = distance
                    next_state = neighbor

        if next_state is None:  # No valid moves, unsolvable
            return "No solution"

        path.append(deepcopy(next_state))
        manhattan_distances.append(manhattan_distance(next_state, sum(goal_state, [])))
        current_state = next_state

    return path, manhattan_distances

# Initial and goal states
initial_state = [
    [1, 2, 3],
    [4, 5, 6],
    [0, 7, 8]
]

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

# Solve the puzzle
solution = solve_8_puzzle(initial_state, goal_state)

# Print the solution path and Manhattan distances
if solution == "No solution":
    print("The puzzle is unsolvable.")
```

```
else:
    path, manhattan_distances = solution
    print("Solution path:")
    for step, state in enumerate(path):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print(f"Manhattan distance: {manhattan_distances[step]}")
        print()
```

output:



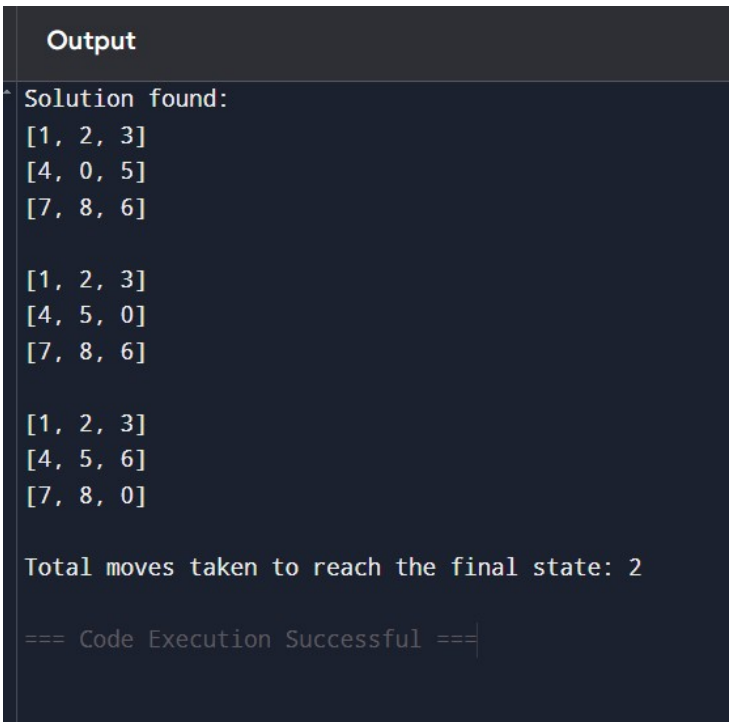
8 puzzle using dfs:

Code:

```
class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.moves = moves
        self.previous = previous
        self.empty_pos = self.find_empty()

    def find_empty(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

    def manhattan_distance(self):
        dist = 0
        for i in range(3):
            for j in range(3):
                tile = self.board[i][j]
                if tile != 0:
                    target_x = (tile - 1) // 3
                    target_y = (tile - 1) % 3
                    dist += abs(i - target_x) + abs(j - target_y)
        return dist

    def generate_moves(self):
        moves = []
        x, y = self.empty_pos
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Directions to move the empty tile

        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
                moves.append(PuzzleState(new_board, self.moves + 1, self))

        return moves

def dfs(start_board, max_depth):
    stack = [PuzzleState(start_board)]
    visited = set()
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

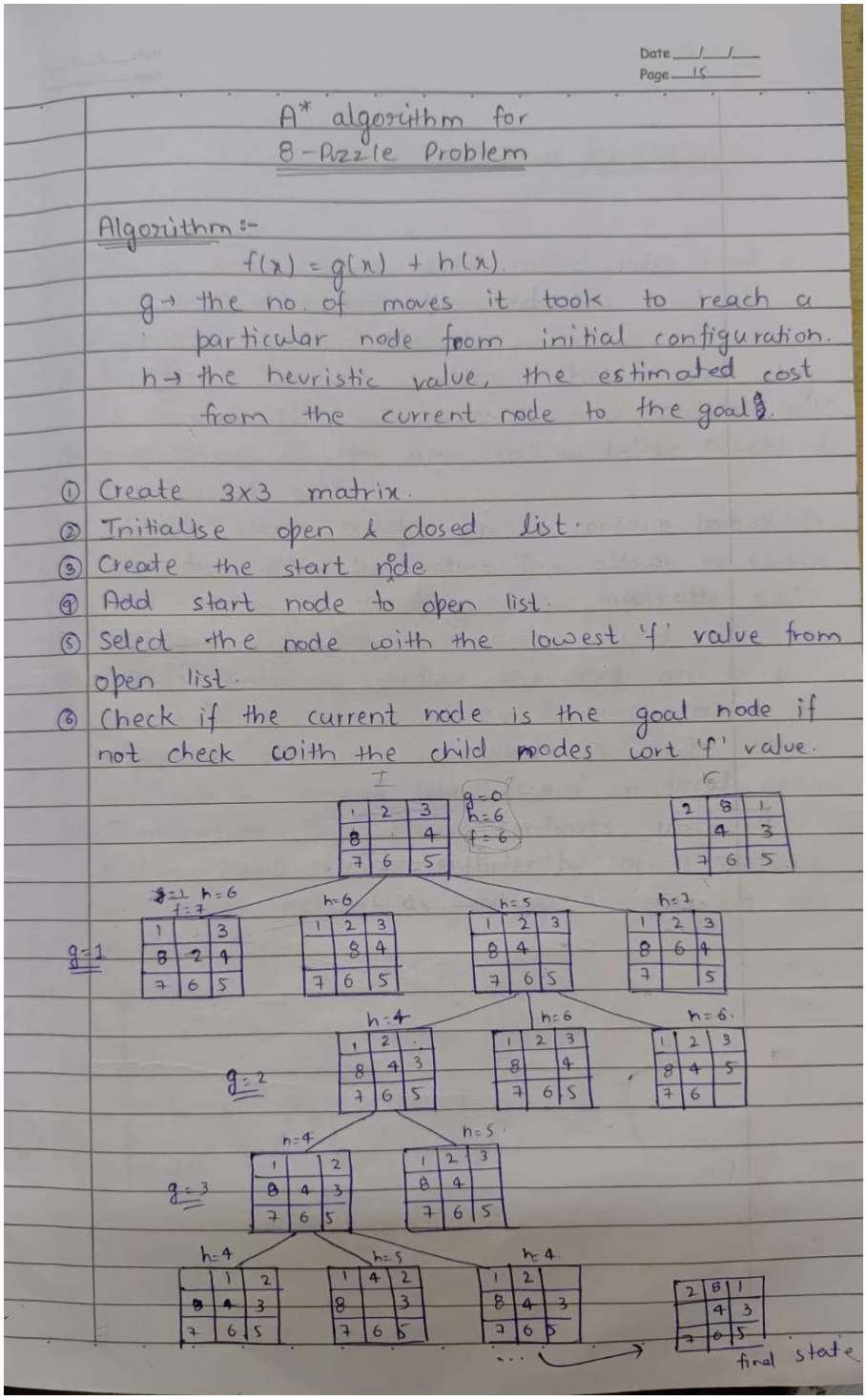
```
    while stack:
        current_state = stack.pop()

        if current_state.board == goal_state:
            return current_state

        visited.add(tuple(map(tuple, current_state.board)))

        if current_state.moves < max_depth:
            for next_state in current_state.generate_moves():
                # Avoid revisiting states and ensure we don't expand too many nodes
                if tuple(map(tuple, next_state.board)) not in visited:
                    stack.append(next_state)

    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for step in reversed(path):
        for row in step:
            print(row)
        print()
    print(f"Total moves taken to reach the final state: {len(path) - 1}")

# Initial state of the puzzle
initial_board = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]
max_depth = 10  # Maximum depth for DFS

solution = dfs(initial_board, max_depth)
if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution found.")
```

output:


Output
Solution found:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Total moves taken to reach the final state: 2
=== Code Execution Successful ===

8 puzzle using A* search:

Algorithm:



A* algorithm for
8-Puzzle Problem

Algorithm :-

$f(x) = g(x) + h(x)$

g → the no. of moves it took to reach a particular node from initial configuration.

h → the heuristic value, the estimated cost from the current node to the goal.

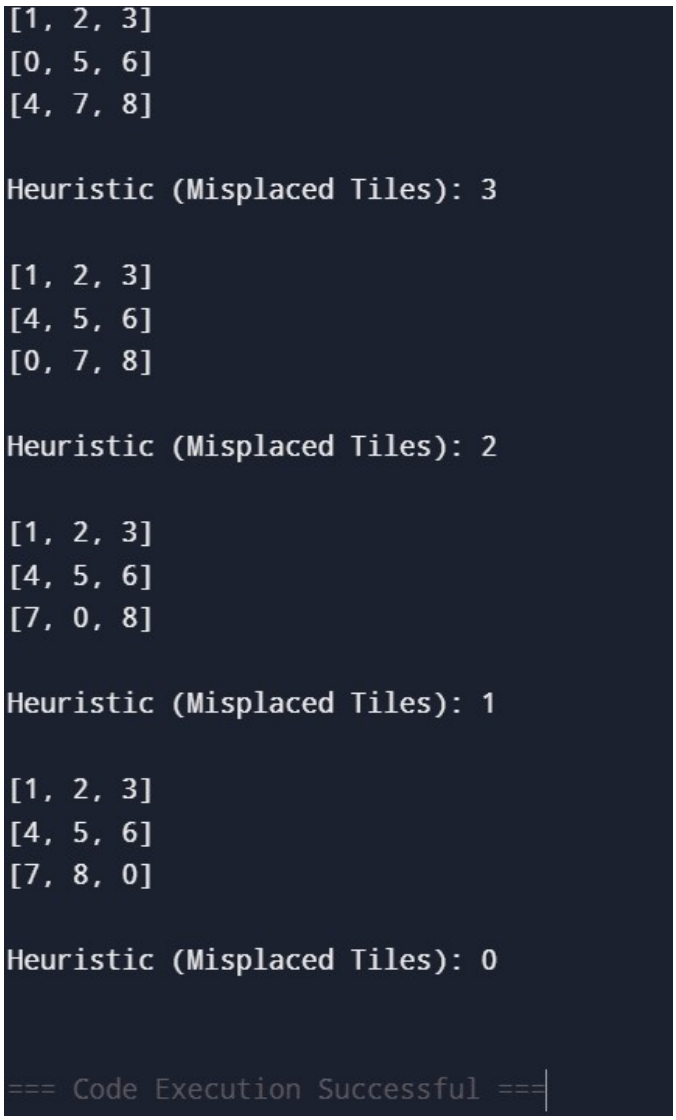
① Create 3x3 matrix.
② Initialise open & closed list.
③ Create the start node.
④ Add start node to open list.
⑤ Select the node with the lowest 'f' value from open list.
⑥ Check if the current node is the goal node if not check with the child nodes sort 'f' value.

Code:

```
import heapq

# Function to print the 8-puzzle grid
def print_puzzle(state):
    for i in range(3):
        print(state[i * 3:(i + 1) * 3])
    print()

# Function to find the position of the blank (0)
def find_blank(state):
    return state.index(0)

# Function to generate possible moves from the current state
def generate_moves(state):
    blank_pos = find_blank(state)
    moves = []

    # Define possible moves (row, column offsets)
    possible_directions = {
        "up": -3,    # Move blank up
        "down": 3,   # Move blank down
        "left": -1,  # Move blank left
        "right": 1   # Move blank right
    }

    # Conditions to avoid invalid moves
    if blank_pos % 3 == 0:  # Left column
        possible_directions.pop("left")
    if blank_pos % 3 == 2:  # Right column
        possible_directions.pop("right")
    if blank_pos < 3:       # Top row
        possible_directions.pop("up")
    if blank_pos > 5:       # Bottom row
        possible_directions.pop("down")

    # Generate new states based on valid moves
    for direction, offset in possible_directions.items():
        new_pos = blank_pos + offset
        new_state = list(state)
        # Swap blank with the new position
        new_state[blank_pos], new_state[new_pos] = new_state[new_pos], new_state[blank_pos]
        moves.append(new_state)

    return moves

# Misplaced tiles heuristic (h)
def misplaced_tiles(state, goal_state):
    return sum(1 for i in range(9) if state[i] != goal_state[i] and state[i] != 0)

# A* Search for 8-puzzle
def a_star_search(start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tiles(start, goal), start))  # (f, state)
    g_costs = {tuple(start): 0}
```

```
    parent_map = {tuple(start): None}

    while open_list:
        _, current_state = heapq.heappop(open_list)

        # If goal state is reached
        if current_state == goal:
            return reconstruct_path(current_state, parent_map)

        # Generate valid moves
        for neighbor in generate_moves(current_state):
            g = g_costs[tuple(current_state)] + 1
            h = misplaced_tiles(neighbor, goal)
            f = g + h

            if tuple(neighbor) not in g_costs or g < g_costs[tuple(neighbor)]:
                g_costs[tuple(neighbor)] = g
                heapq.heappush(open_list, (f, neighbor))
                parent_map[tuple(neighbor)] = current_state

    return "No solution"

# Reconstruct the solution path
def reconstruct_path(goal_state, parent_map):
    path = []
    while goal_state:
        path.append(goal_state)
        goal_state = parent_map[tuple(goal_state)]
    return path[::-1]

# Example usage
start_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]  # Initial state
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]   # Goal state

print("Initial State:")
print_puzzle(start_state)

solution_path = a_star_search(start_state, goal_state)

if solution_path != "No solution":
    print("Solution path found:")
    for state in solution_path:
        print_puzzle(state)
        # Calculate and display the heuristic value for each state
        print(f"Heuristic (Misplaced Tiles): {misplaced_tiles(state, goal_state)}\n")
else:
    print("No solution found.")
```

Output:


[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
Heuristic (Misplaced Tiles): 3
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Heuristic (Misplaced Tiles): 2
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Heuristic (Misplaced Tiles): 1
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Heuristic (Misplaced Tiles): 0
=== Code Execution Successful ===

Simulated Annealing

Algorithm :

Date 22/10/24
Page 17

LAB- 5

SIMULATED ANNEALING ALGORITHM

① Initialisation :- Start with initial value and a high temp value. Temperature represents level of exploration in sol^n space.

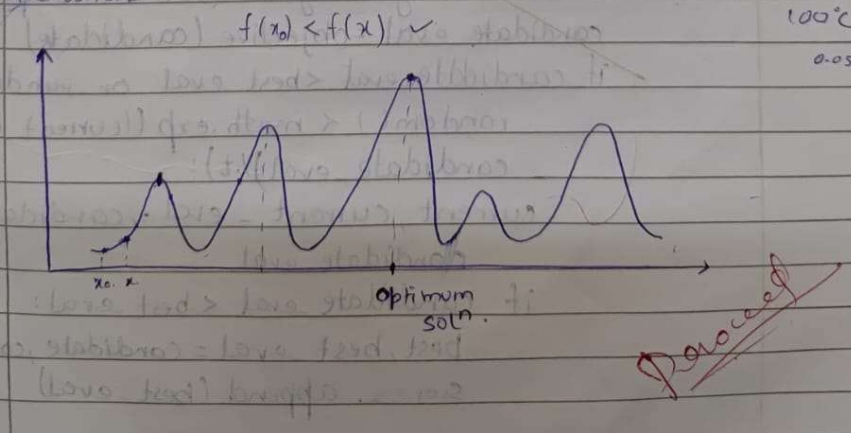
② Iteration :- In each iteration, make a small chng to current sol^n to create a new candidate sol^n.

③ Evaluation :- If the new sol^n is better, accept it as current sol^n.

• If it's worse, accept with a certain probability that decr. with temperature. This allows to escape local optima by accepting less favourable sol^n at high temperature.

④ Cooling :- Gradually reduce the temp acc. to a predefined cooling schedule. As temp lowers, the algorithm becomes less likely to accept worse sol^n, which narrows the focus on local optima.

⑤ Termination :- The process continues until the system cools down sufficiently or a stopping criteria is met, such as reaching a max no. of iterations.



```
import math
import random
def objective_function(x):
    return 10*len(x) + sum([xi**2 - 10*math.cos(
        2*math.pi*xi)) for xi in x])
def get_neighbor(x, step_size=0.1):
    neighbour = x[:]
    index = random.randint(0, len(x)-1)
    neighbor[index] += random.uniform(-step_size,
        step_size)
    return neighbor
def simulated_annealing(objective, bounds, n_iterations,
    step_size, temp):
    best = [random.uniform(bound[0], bound[1]) for
        bound in bounds]
    best_eval = objective(best)
    current, current_eval = best, best_eval
    scores = [best_eval]

    for i in range(n_iterations):
        t = temp / float(i+1):
        candidate = get_neighbor(current, step_size)
        candidate_eval = objective(candidate)
        if candiddte_eval < best_eval or random.
            random() < math.exp((current_eval -
            candidate_eval)/t):
            current, current_eval = candidate,
                candidate_eval
            if candidate_eval < best_eval:
                best, best_eval = candidate, candidate
                scores.append(best_eval)
```

Date / /
Page 19

```
    if i%.100 == 0:
        print(f"Iteration {i}, Temperature {t:3f}
        , Best Evaluation {best_eval:.5f}")
return best, best_eval, scores.
bounds = [[-5.0, 5.0) for _ in range(2)]
n_iterations = 1000
step_size = 0.1
temp = 10
best, score, scores = simulated_annealing (objective_
    function, bounds, n_iterations, step_size, temp)
print(f'Best solution: {best}')
print(f'Best score: {score}')
```

OUTPUT:-

```
Iteration 0, Temperature 10.000, Best evaluation 46.19632
    100,        0.99,        7.98141
    200,        0.050,       7.96350
    300,        0.033,       7.96350
     :
    900,        0.011,       7.96077
Best solution: [1.9913780032716961, 1.9917676275892 9]
Best score: 7.960769607 57.
```

8/22/24

Code:

```
importrandom
import math
defcount_attacks(queen_positions):
    attack_count = 0
    size=len(queen_positions)
    for i in range(size):
        forjinrange(i+1,size):
            ifqueen_positions[i]==queen_positions[j]:
                attack_count += 1
            ifabs(queen_positions[i]-queen_positions[j])==abs(i-j): attack_count
                += 1
    returnattack_count
defgenerate_random_move(current_positions):
    new_state = current_positions[:]
    column_to_change=random.randint(0,len(current_positions)-1)
    new_row_position=random.randint(0,len(current_positions)-1)
    new_state[column_to_change] = new_row_position
    returnnew_state
def annealing_search(board_size, initial_configuration):
    current_positions = initial_configuration[:]
    current_attack_count=count_attacks(current_positions)
    temp = 1000
    min_temp=0.0001
    cooling_factor=0.99
    step_count = 0
    visited_states=set()
    whiletemp>min_tempandcurrent_attack_count>0:
        step_count += 1

        #ifstep_count%150== 0:
            #print("Step",step_count,":Attacks=",current_attack_count)

        new_positions = generate_random_move(current_positions)

        new_positions_tuple = tuple(new_positions)

        ifnew_positions_tupleinvisited_states:
            continue
```

```
        visited_states.add(new_positions_tuple)

        new_attack_count=count_attacks(new_positions)

        energy_difference=new_attack_count-current_attack_count

        ifenergy_difference<0orrandom.random()<math.exp(-energy_difference/temp): current_positions,
            current_attack_count = new_positions, new_attack_count
        temp*=cooling_factor
        ifcurrent_attack_count==0:
            print("Solutionfoundafter",step_count,"steps!")
            break
    return current_positions, current_attack_count
board_size=int(input("Enterthesizeoftheboard(N):"))
initial_input = input("Enter the initial configuration: ")
initial_queen_positions=[int(pos)forposininitial_input.strip('[]').split(',')]

iflen(initial_queen_positions)!=board_size:
    print("Error:Theinitialconfigurationmustcontainexactly",board_size,"integers.") else:
    solution,conflicts=annealing_search(board_size,initial_queen_positions)

    if conflicts == 0:
        print("Solutionfound!")
        print("Finalboardconfiguration:",solution)
    else:
        print("Nosolutionfound.Finalconflictcount:",conflicts)
```

Output:

```
Enter the size of the board (N): 8
Enter the initial configuration: 4,5,6,3,4,5,6,5
Solution found after 1093 steps!
Solution found!
Final board configuration: [5, 3, 6, 0, 2, 4, 1, 7]
```

8-queens using hill climbing :

Algorithm:

LAB → 6

8-QUEENS PROBLEM

Hill climbing Algorithm:-

① Start with random state / random configuration of board.

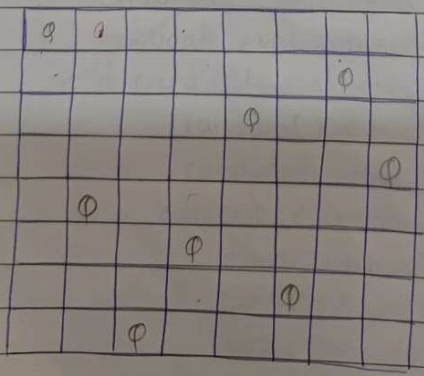
② Scan all possible neighbours of current state and jump to the neighbour with highest objective value. If no other state with higher objective exists, then directly jump to any random neighbour.

③ Repeat step 2, until a state whose objective is higher than all it's neighbour objective is found.

④ State found after the search will be either local optimum or global optimum.

⑤ The global optimum 0 since it is the min. no. of pairs of queens that can attack each other. The random restart has a higher chance of achieving global optimum bcoz the problem has high no. of local optimum and random neighbour is faster than random restart.

⑥ Output the state and return.



Code:

```
import random

def calculate_conflicts(board):
    """Calculates the number of pairs of queens attacking each other."""
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def get_neighbors(board):
    """Generates neighboring boards by swapping two queens."""
    neighbors = []
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            new_board = board[:]
            new_board[i], new_board[j] = new_board[j], new_board[i]  # Swap queens
            neighbors.append(new_board)
    return neighbors

def hill_climbing(board):
    """Solves the N-Queens problem using the Hill Climbing algorithm."""
    current_conflicts = calculate_conflicts(board)
    print(f"Initial board: {board} with {current_conflicts} conflicts")

    while True:
        neighbors = get_neighbors(board)
        next_board = None
        next_conflicts = current_conflicts

        for neighbor in neighbors:
            conflicts = calculate_conflicts(neighbor)
            if conflicts < next_conflicts:
                next_conflicts = conflicts
                next_board = neighbor

        if next_conflicts >= current_conflicts:  # If no improvement, stop
            break

        board = next_board
        current_conflicts = next_conflicts
        print(f"Intermediate board: {board} with {current_conflicts} conflicts")
```

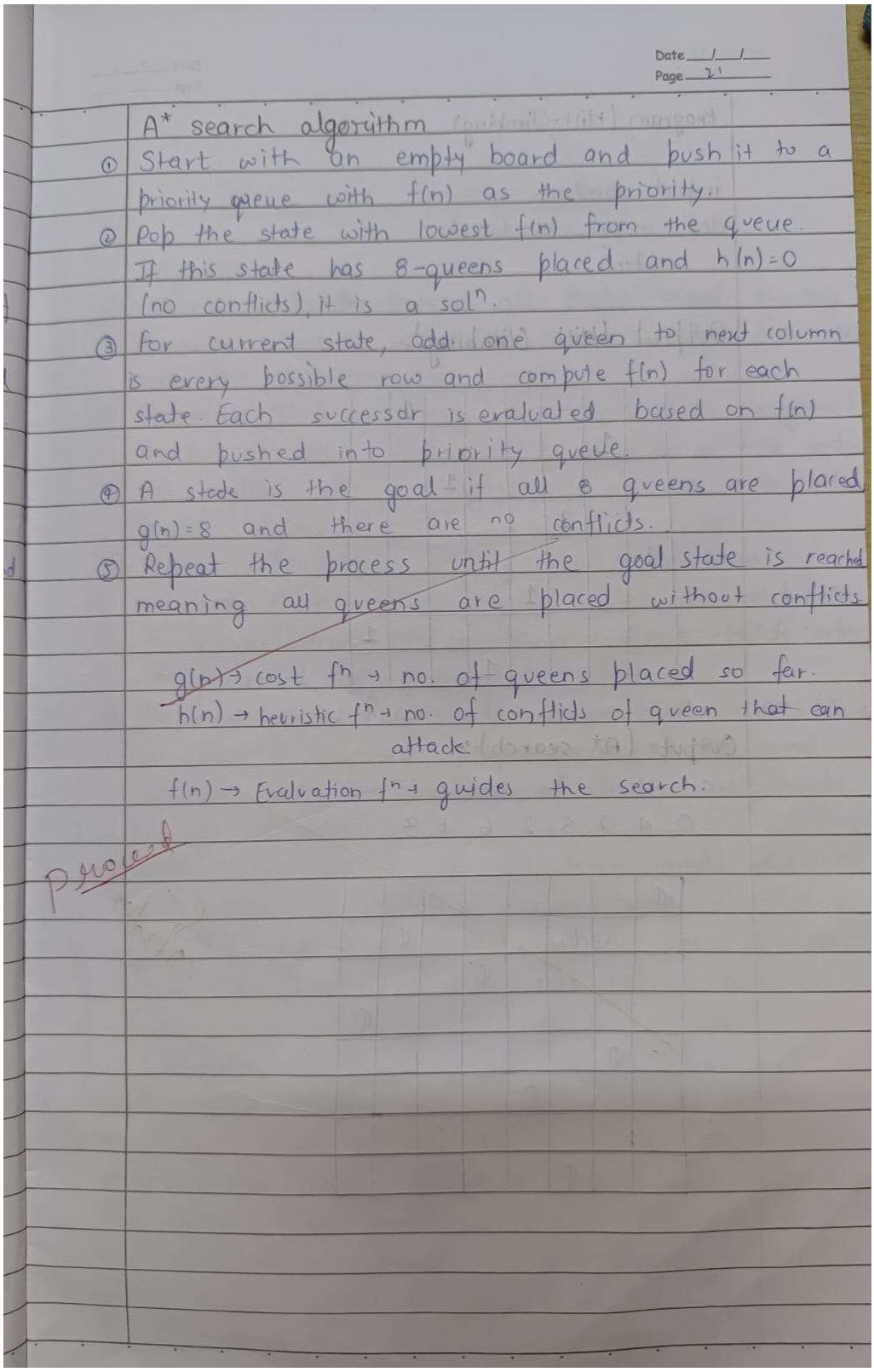
```
    return board, current_conflicts

# Input and solving
try:
    n = int(input("Enter the number of queens (size of the board): "))
    if n <= 0:
        raise ValueError("The number of queens must be a positive integer.")

    board = []
    for i in range(n):
        row = int(input(f"Enter the row index for queen {i + 1} (0 to {n - 1}): "))
        if row < 0 or row >= n:
            raise ValueError("Invalid row index. Must be within the range 0 to n-1.")
        board.append(row)

    solution, conflicts = hill_climbing(board)

    if conflicts == 0:
        print("Solution found:")
        print(solution)
    else:
        print("No solution found, best configuration with conflicts:")
        print(solution, "with", conflicts, "conflicts.")

except ValueError as e:
    print(f"Invalid input: {e}")
```

Output:

```
Enter the number of queens (size of the board): 8
Enter the row index for queen 1 (0 to 7): 2
Enter the row index for queen 2 (0 to 7): 7
Enter the row index for queen 3 (0 to 7): 5
Enter the row index for queen 4 (0 to 7): 1
Enter the row index for queen 5 (0 to 7): 0
Enter the row index for queen 6 (0 to 7): 3
Enter the row index for queen 7 (0 to 7): 4
Enter the row index for queen 8 (0 to 7): 6
Initial board: [2, 7, 5, 1, 0, 3, 4, 6] with 5 conflicts
Intermediate board: [2, 7, 5, 3, 0, 1, 4, 6] with 2 conflicts
Intermediate board: [1, 7, 5, 3, 0, 2, 4, 6] with 1 conflicts
No solution found, best configuration with conflicts:
[1, 7, 5, 3, 0, 2, 4, 6] with 1 conflicts.

=== Code Execution Successful ===
```

8 queens using A* search :

Algorithm :



A* search algorithm

① Start with an empty board and push it to a priority queue with f(n) as the priority.

② Pop the state with lowest f(n) from the queue. If this state has 8-queens placed and h(n)=0 (no conflicts), it is a sol^n.

③ For current state, add one queen to next column in every possible row and compute f(n) for each state. Each successor is evaluated based on f(n) and pushed into priority queue.

④ A state is the goal if all 8 queens are placed, g(n)=8 and there are no conflicts.

⑤ Repeat the process until the goal state is reached meaning all queens are placed without conflicts.

g(n) → cost fn → no. of queens placed so far.

h(n) → heuristic fn → no. of conflicts of queen that can attack.

f(n) → Evaluation fn → guides the search.

Code:

```
import heapq

# Function to evaluate the heuristic (number of attacking pairs of queens)
def evaluate_heuristic(state):
    attacks = 0
    for i in range(8):
        for j in range(i + 1, 8):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

# Function to generate neighbors (states) by moving one queen in each column
def generate_neighbors(state):
    neighbors = []
    for i in range(8):
        for j in range(8):
            if state[i] != j:
                new_state = state[:]
                new_state[i] = j
                neighbors.append(new_state)
    return neighbors

# Function to print the board in a readable format
def print_board(state):
    board = []
    for row in state:
        row_rep = ["."] * 8
        row_rep[row] = "Q"
        board.append(" ".join(row_rep))
    for row in board:
        print(row)
    print()

# A* Search algorithm for 8-Queens
def a_star_8_queens():
    initial_state = [0, 1, 2, 3, 4, 5, 6, 7]  # Random initial state, queens in columns 0 to 7

    # Priority queue for A* search (f(n), g(n), state)
    pq = []
    heapq.heappush(pq, (evaluate_heuristic(initial_state), 0, initial_state))

    visited = set()  # To avoid revisiting states

    while pq:
        # Pop the state with the lowest f(n) = g(n) + h(n)
        h_value, g_value, current_state = heapq.heappop(pq)
```

```
        print(f"Evaluating state with f(n)={g_value + h_value} (g(n)={g_value}, h(n)={h_value})")
        print_board(current_state)

        # If we find a solution where no queens are attacking each other (h(n) == 0)
        if h_value == 0:
            print("Solution found!")
            return current_state

        # Generate neighbors
        neighbors = generate_neighbors(current_state)

        for neighbor in neighbors:
            if tuple(neighbor) not in visited:
                visited.add(tuple(neighbor))
                h_neighbor = evaluate_heuristic(neighbor)
                g_neighbor = g_value + 1  # Cost is simply the depth or number of moves
                f_neighbor = g_neighbor + h_neighbor
                heapq.heappush(pq, (h_neighbor, g_neighbor, neighbor))

    print("No solution found.")
    return None

# Run the A* algorithm
solution = a_star_8_queens()

if solution:
    print("Final solution:")
    print_board(solution)
else:
    print("No solution found.")
```

output:

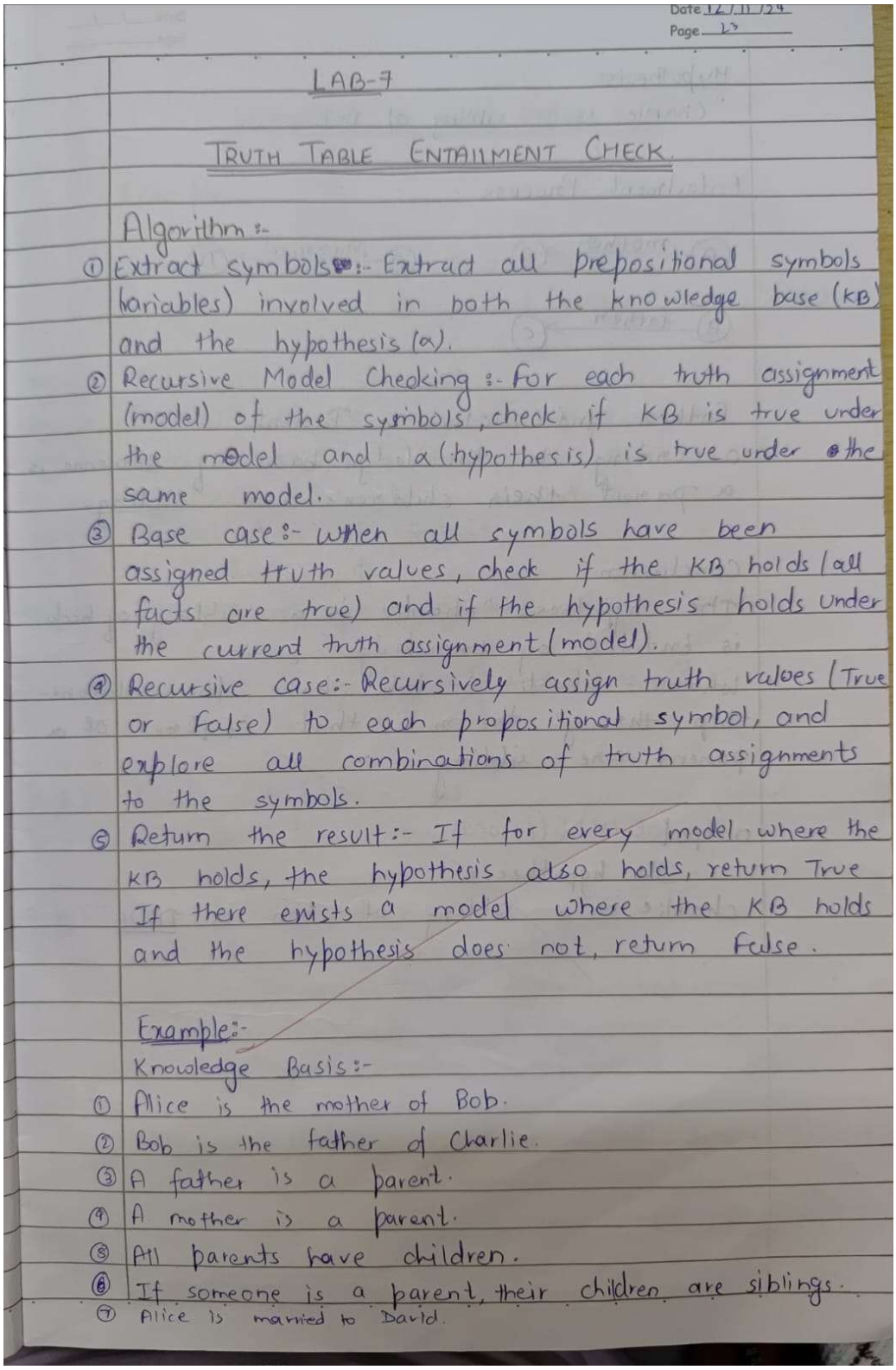
```
. . . . . Q . .
. . Q . . . . .

Evaluating state with f(n)=10 (g(n)=10, h(n)=0)
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .

Solution found!
Final solution:
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
```

Output

```
Evaluating state with f(n)=28 (g(n)=0, h(n)=28)
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q

Evaluating state with f(n)=23 (g(n)=1, h(n)=22)
Q . . . . . . .
Q . . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q

Evaluating state with f(n)=18 (g(n)=2, h(n)=16)
Q . . . . . . .
Q . . . . . . .
. . Q . . . . .
. . . Q . . . .
. Q . . . . . .
```

Truth table entailment check:

Algorithm:



Date 12/11/24
Page 23

LAB-7

TRUTH TABLE ENTAILMENT CHECK.

Algorithm :-

① Extract symbols :- Extract all propositional symbols (variables) involved in both the knowledge base (KB) and the hypothesis (α).

② Recursive Model Checking :- For each truth assignment (model) of the symbols, check if KB is true under the model and α (hypothesis) is true under the same model.

③ Base case :- When all symbols have been assigned truth values, check if the KB holds (all facts are true) and if the hypothesis holds under the current truth assignment (model).

④ Recursive case :- Recursively assign truth values (True or False) to each propositional symbol, and explore all combinations of truth assignments to the symbols.

⑤ Return the result :- If for every model where the KB holds, the hypothesis also holds, return True If there exists a model where the KB holds and the hypothesis does not, return False.

Example :-

Knowledge Basis :-

① Alice is the mother of Bob.
② Bob is the father of Charlie.
③ A father is a parent.
④ A mother is a parent.
⑤ All parents have children.
⑥ If someone is a parent, their children are siblings.
⑦ Alice is married to David.

Date __/__/__
Page _____

Hypothesis:-
"Charlie is a sibling of Bob."

Entailment Process :

(A) mother → (B) (A) Married → (D)

(B) father → (C)

Father and Mother in Parent according to knowledge basis if someone is a Parent their children's are Siblings

Conclusion:
The hypothesis "charlie is a sibling of bob" is Entailed by knowledge basis where charli & bob share the relationship through the rule that children of a parent are sibling

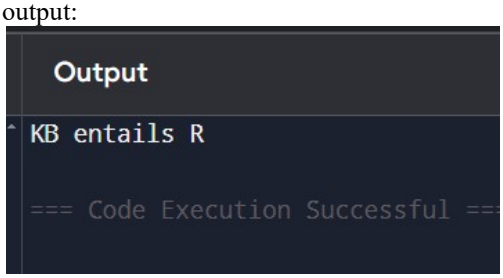
output :
The hypothesis :
charlie is a sibling of bob is TRUE

Code:

```
import random
import itertools
import math

def eval_formula(formula, assignment):
    """Evaluates a logical formula given a truth assignment."""
    # Replace logical operators in the formula to Python equivalents
    formula = formula.replace('and', 'and').replace('or', 'or').replace('not', 'not')
    formula = formula.replace('→', 'or not')  # Implication A → B is equivalent to (not A or B)
    formula = formula.replace('↔', '==')  # Equivalence A ↔ B is equivalent to (A == B)

    # Create a dictionary environment with variable assignments
    env = {var: value for var, value in zip(assignment.keys(), assignment.values())}
    return eval(formula, {}, env)

def generate_initial_state(variables):
    """Generates a random initial truth assignment for the given variables."""
    return {var: random.choice([True, False]) for var in variables}

def entails(KB, alpha):
    """Checks if the knowledge base (KB) entails the formula alpha."""
    # Find all unique variables in KB and alpha
    variables = set(''.join([ch for ch in ''.join(KB + [alpha]) if ch.isalpha()]))

    # Generate all possible truth assignments for the variables
    truth_assignments = list(itertools.product([True, False], repeat=len(variables)))
    var_list = list(variables)

    for assignment in truth_assignments:
        # Map the truth assignment to each variable in the assignment
        assignment_dict = dict(zip(var_list, assignment))

        # Combine all KB formulas with AND and evaluate
        kb_combined = all(eval_formula(formula, assignment_dict) for formula in KB)
        alpha_true = eval_formula(alpha, assignment_dict)

        # If KB is true and alpha is false for any assignment, KB does not entail alpha
        if kb_combined and not alpha_true:
            return False

    # If we reach here, it means KB entails alpha
    return True

# Example usage:
KB = ["not Q or P", "not P or (not Q)", "Q or R"]
alpha = "R"
```

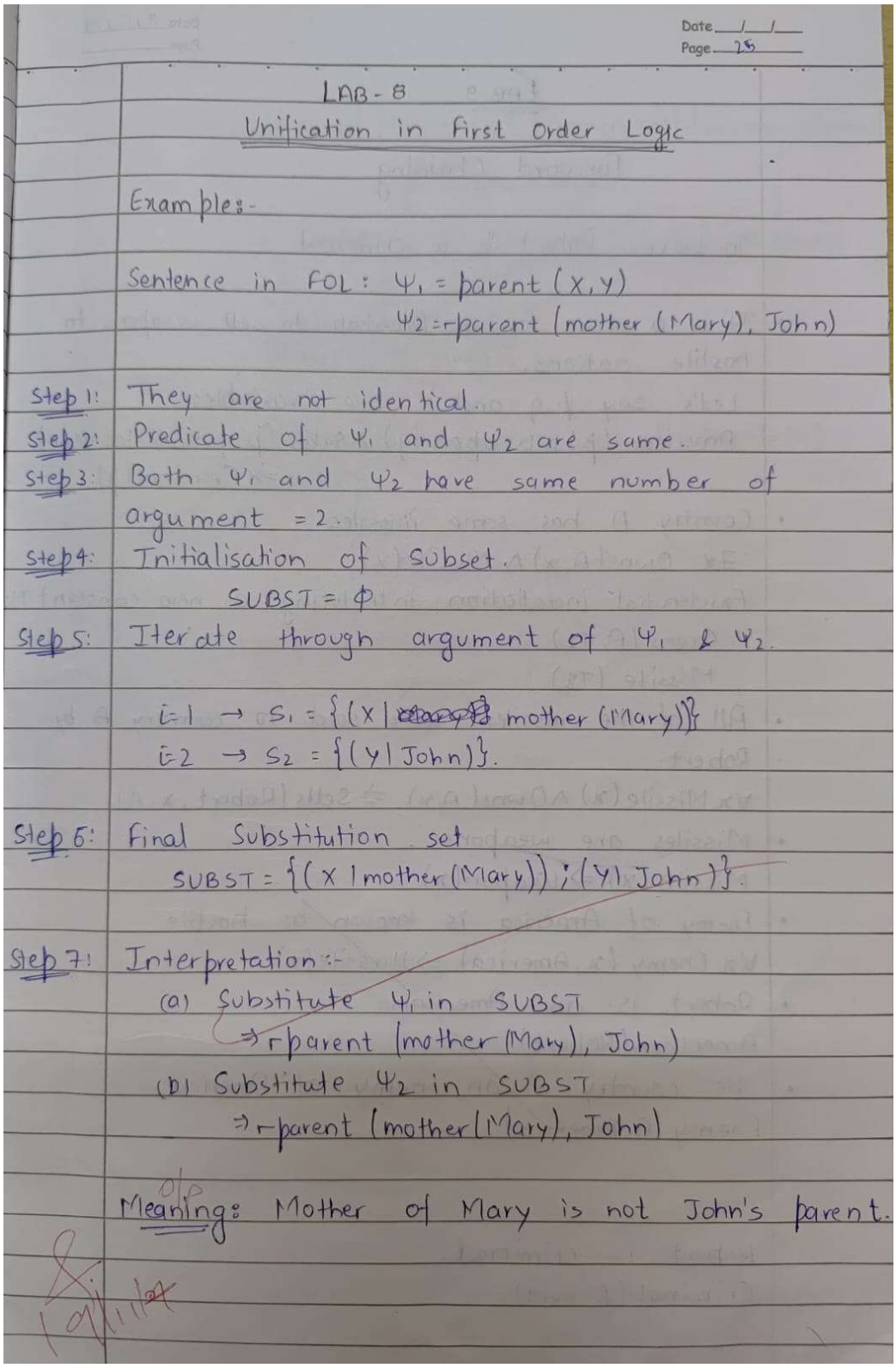
```
# Check if KB entails alpha
if entails(KB, alpha):
    print("KB entails R")
else:
    print("KB does not entail R")
```

output:



Unification in first order logic :

Algorithm :



Date / /
Page 25

LAB - 8

Unification in First Order Logic

Examples:-

Sentence in FOL: Ψ_1 = parent (X, Y)

Ψ_2 = parent (mother (Mary), John)

Step 1: They are not identical.

Step 2: Predicate of Ψ_1 and Ψ_2 are same.

Step 3: Both Ψ_1 and Ψ_2 have same number of argument = 2.

Step 4: Initialisation of Subset.

SUBST = ϕ

Step 5: Iterate through argument of Ψ_1 & Ψ_2.

i=1 → S_1 = {(X | mother (Mary))}

i=2 → S_2 = {(Y | John)}.

Step 6: Final Substitution set

SUBST = {(X | mother (Mary)) ; (Y | John)}.

Step 7: Interpretation :-

(a) Substitute Ψ_1 in SUBST

⇒ parent (mother (Mary), John)

(b) Substitute Ψ_2 in SUBST

⇒ parent (mother (Mary), John)

Meaning: Mother of Mary is not John's parent.

Code:

```
import random
import itertools
import math

def occurs_check(var, term):
    """Checks if a variable occurs in a term."""
    if isinstance(term, str):  # Term is a constant
        return False
    elif isinstance(term, tuple):  # Term is a function (represented as a tuple)
        # Recursively check the function arguments
        if term[0] == var:
            return True
        return any(occurs_check(var, arg) for arg in term[1:])
    return False

def unify(psi1, psi2, subst=None):
    """Unify two terms psi1 and psi2 with the current substitution."""
    if subst is None:
        subst = {}

    # Step 1: If either term is a variable or constant
    if isinstance(psi1, str):  # psi1 is a variable
        if psi1 == psi2:  # Identical variables
            return subst
        elif psi1 in subst:  # psi1 already has a substitution
            return unify(subst[psi1], psi2, subst)
        elif occurs_check(psi1, psi2):  # Occurs check
            return "FAILURE"
        else:
            subst[psi1] = psi2  # Create a new substitution
            return subst

    elif isinstance(psi2, str):  # psi2 is a variable
        if psi2 == psi1:  # Identical variables
            return subst
        elif psi2 in subst:  # psi2 already has a substitution
            return unify(psi1, subst[psi2], subst)
        elif occurs_check(psi2, psi1):  # Occurs check
            return "FAILURE"
        else:
            subst[psi2] = psi1  # Create a new substitution
            return subst

    # Step 2: Check if the initial predicates symbols match
    if isinstance(psi1, tuple) and isinstance(psi2, tuple):
        if psi1[0] != psi2[0]:  # Predicate symbols don't match
```

```
            return "FAILURE"

    # Step 3: Check if they have the same number of arguments
    if isinstance(psi1, tuple) and isinstance(psi2, tuple):
        if len(psi1) != len(psi2):  # Different number of arguments
            return "FAILURE"

    # Step 4: Initialize the substitution set (already initialized as `subst`)
    # Step 5: Iterate through the arguments of psi1 and psi2
    if isinstance(psi1, tuple) and isinstance(psi2, tuple):
        for arg1, arg2 in zip(psi1[1:], psi2[1:]):  # Recursively unify the arguments
            result = unify(arg1, arg2, subst)
            if result == "FAILURE":
                return "FAILURE"
            elif result != subst:
                subst = result  # Update the substitution set

    # Step 6: Return the final substitution set
    return subst

# Helper function to display the substitution in a readable format
def print_substitution(subst):
    if subst == "FAILURE":
        print("FAILURE")
    else:
        for var, val in subst.items():
            print(f"{var} -> {val}")

# Helper function to parse the user input into the required format
def parse_input(input_str):
    """Parse the input string into a tuple representing the term."""
    input_str = input_str.strip()
    if '(' in input_str and ')' in input_str:
        # Extract the predicate and arguments
        predicate, args = input_str.split('(', 1)
        args = args.rstrip(')').split(',')
        args = [parse_input(arg.strip()) if '(' in arg else arg.strip() for arg in args]
        return (predicate.strip(), *args)
    else:
        # Single term, assume it's a constant or a variable
        return input_str.strip()

# Main function to handle user input and unification
def main():
    print("Please enter the first term (e.g., f(X, g(Y))):")
    term1 = input().strip()

    print("Please enter the second term (e.g., f(a, g(b))):")
```

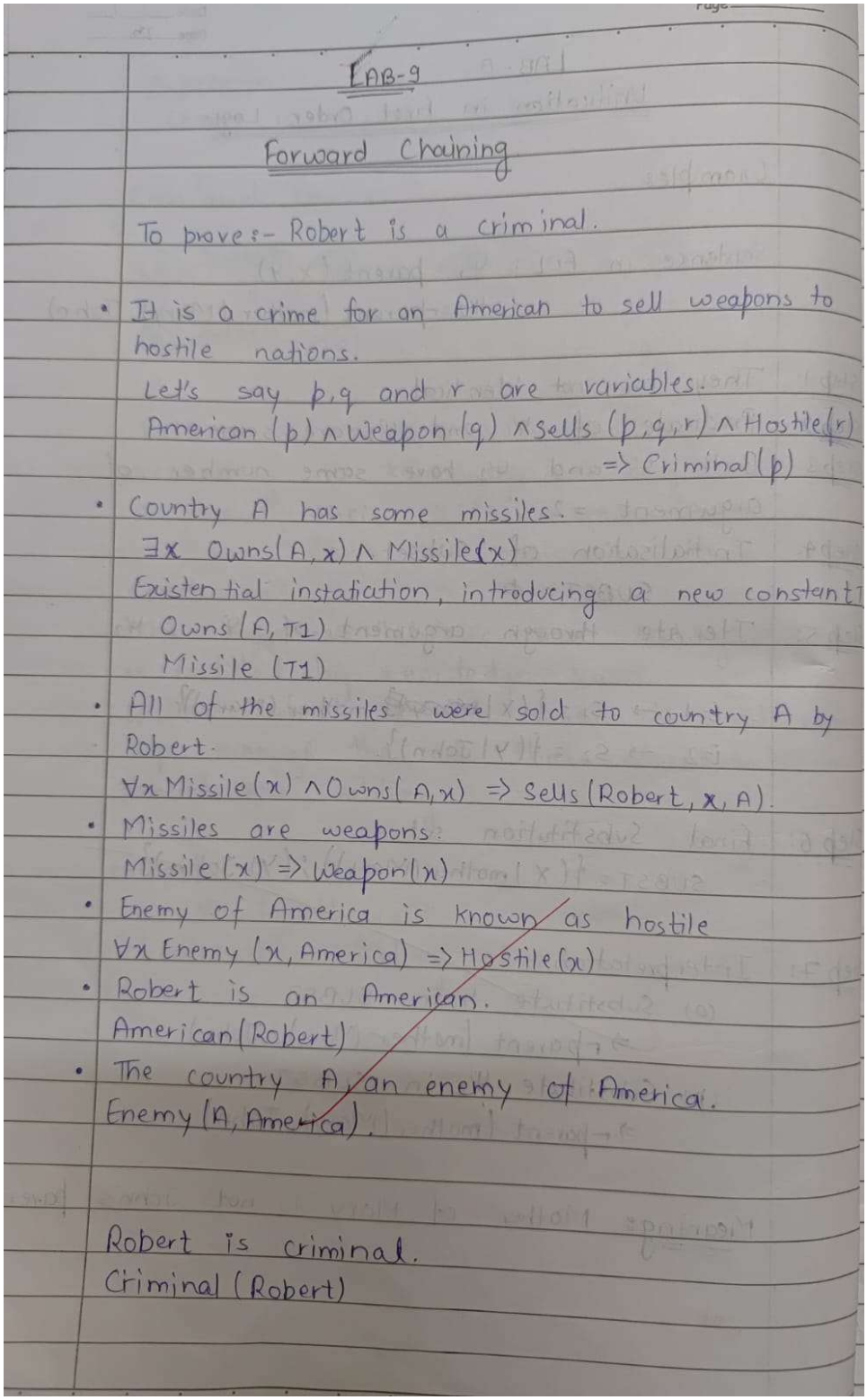
```
    term2 = input().strip()

    # Parse the user input into structured terms
    psi1 = parse_input(term1)
    psi2 = parse_input(term2)

    # Unify the terms and print the result
    substitution = unify(psi1, psi2)
    print_substitution(substitution)

# Run the main function
if __name__ == "__main__":
    main()
```

Output:

```
Please enter the first term (e.g., f(X, g(Y))):
American(robert,x)
Please enter the second term (e.g., f(a, g(b))):
American(y,enemy)
robert -> y
x -> enemy
```

Forward chaining:

Algorithm :



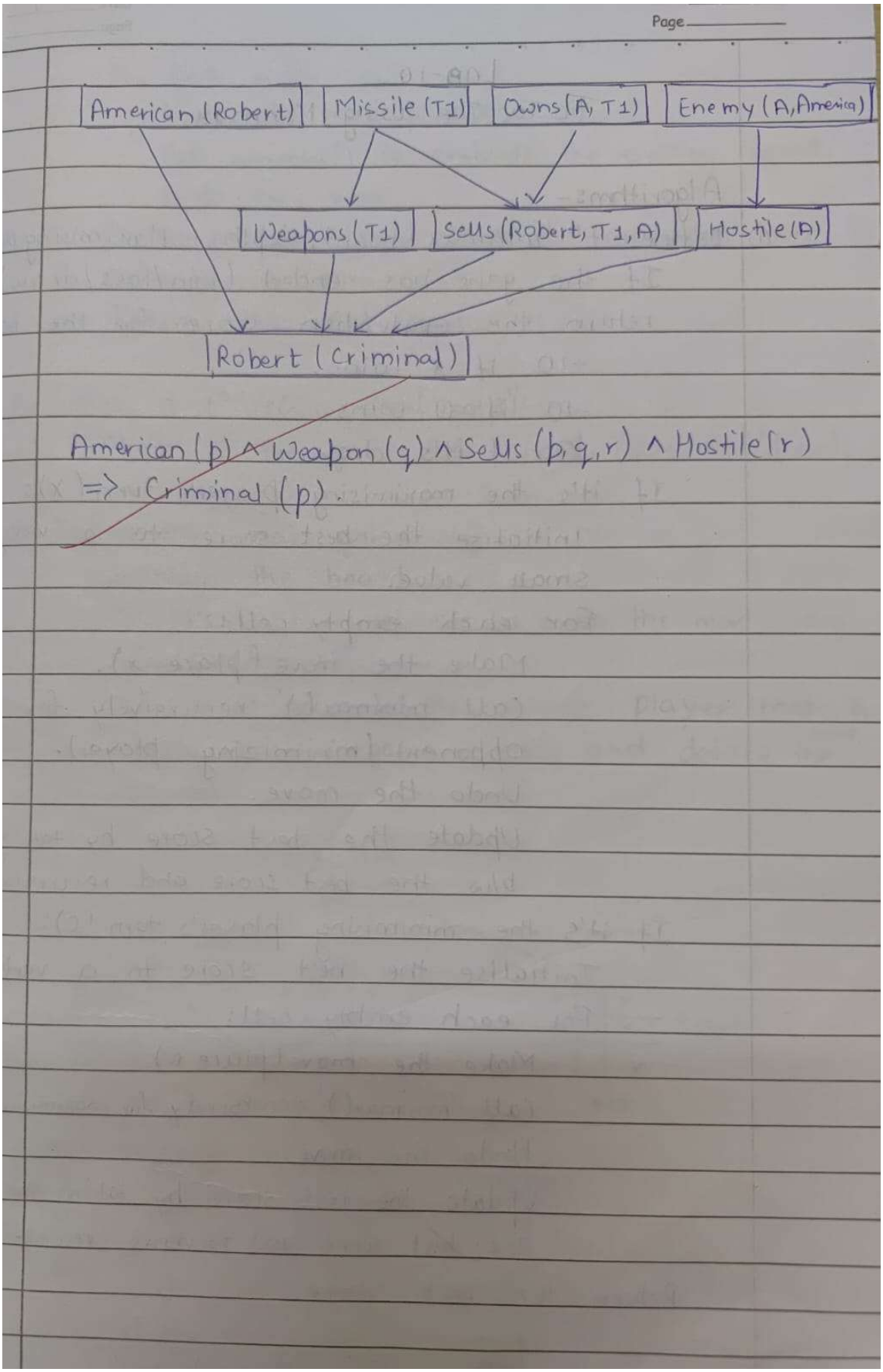
LAB-9

Forward Chaining

To prove:- Robert is a criminal.

- It is a crime for an American to sell weapons to hostile nations.
  Let's say p,q and r are variables.
  American (p) ∧ Weapon (q) ∧ Sells (p,q,r) ∧ Hostile(r) => Criminal(p)
- Country A has some missiles.
  ∃x Owns(A,x) ∧ Missile(x)
  Existential instatiation, introducing a new constant T1
  Owns (A, T1)
  Missile (T1)
- All of the missiles were sold to country A by Robert.
  ∀x Missile(x) ∧ Owns(A,x) => Sells (Robert, x, A).
- Missiles are weapons.
  Missile (x) => Weapon(x)
- Enemy of America is known as hostile
  ∀x Enemy (x, America) => Hostile(x)
- Robert is an American.
  American (Robert)
- The country A, an enemy of America.
  Enemy (A, America).

Robert is criminal.
Criminal (Robert)



American (p) ∧ Weapon (q) ∧ Sells (p, q, r) ∧ Hostile (r)
=> Criminal (p).

Code:

```
class ForwardChaining:
    def __init__(self, facts, rules):
        """
        Initialize the Forward Chaining algorithm with facts and rules.
        :param facts: Set of known facts (initial facts).
        :param rules: List of rules where each rule is a tuple (premise, conclusion).
        """
        self.facts = set(facts)
        self.rules = rules
        self.inferred_facts = set(facts)  # Set of facts derived during the process

    def apply_rule(self, rule):
        """
        Applies a rule to derive new facts from existing facts.
        :param rule: A rule represented as (premise, conclusion).
        :return: True if a new fact is derived, False otherwise.
        """
        premise, conclusion = rule
        premise_facts = set(premise.split(','))  # Split the premise into individual facts

        # Check if the premise of the rule is fully satisfied by current facts
        if premise_facts.issubset(self.facts):  # Ensure all premises are in facts
            if conclusion not in self.facts:
                print(f"Inferred new fact: {conclusion}")
                self.facts.add(conclusion)  # Add the conclusion to the set of facts
                return True
        return False

    def forward_chaining(self):
        """
        Applies forward chaining to derive new facts until no more facts can be derived.
        """
        new_inference = True
        while new_inference:
            new_inference = False
            # Go through all the rules and try to apply them
            for rule in self.rules:
                if self.apply_rule(rule):
                    new_inference = True
            if new_inference:
                print(f"Current facts: {self.facts}")
        print("Forward Chaining completed.")

    def is_goal_reached(self, goal):
        """
```

```
        Checks if the goal has been reached (i.e., if the goal is in the facts).
        :param goal: The goal fact to check for.
        :return: True if the goal is in the facts, otherwise False.
        """
        return goal in self.facts


def main():
    print("Forward Chaining System")

    # Define the initial facts
    facts = {
        "american(p)",
        "weapon(q)",
        "sells(p,q,r)",
        "hostile(r)",
        "american(robert)",
        "enemy(a,america)"
    }

    # Define the rules (premise -> conclusion)
    rules = [
        ("american(p),weapon(q),sells(p,q,r),hostile(r)", "criminal(p)"),  # Rule 1
        ("owns(a,x),missile(x)", "sells(robert,x,a)"),  # Rule 2
        ("missile(x)", "weapon(x)"),  # Rule 3
        ("enemy(a,america)", "hostile(a)")  # Rule 4
    ]

    # Create an instance of ForwardChaining with the facts and rules
    fc = ForwardChaining(facts, rules)

    # Perform forward chaining to infer new facts
    fc.forward_chaining()

    # Define the goal (fact you want to check)
    goal = "criminal(p)"

    # Check if the goal is reached
    if fc.is_goal_reached(goal):
        print(f"The goal '{goal}' is reached!")
    else:
        print(f"The goal '{goal}' is not reached.")


# Run the main function
if __name__ == "__main__":
    main()
```

Output:

```
Forward Chaining System
Inferred new fact: criminal(p)
Current facts: {'american(robert)', 'sells(p,q,r)', 'hostile(r)', 'american(p)', 'weapon(q
Forward Chaining completed.
The goal 'criminal(p)' is reached!
```

Program9

KnowledgeBaseconsistingofFirstOrderLogicStatementsandproofusingResolution:

Algorithm:

Code:

```
importtime
start_time=time.time()
import re
import itertools
importcollections
import copy
import queue

p=open("input.txt","r")
data=list()
data1=p.readlines()
count=0

n=int(data1[0])
queries=list()
for i in range(1,n+1):
    queries.append(data1[i].rstrip())
k=int(data1[n+1])
kbbefore=list()

def CNF(sentence):
    temp=re.split("=>",sentence)
    temp1=temp[0].split('&')
    foriinrange(0,len(temp1)): if
        temp1[i][0]=='~':
            temp1[i]=temp1[i][1:]
        else:
            temp1[i]='~'+temp1[i]
    temp2='|'.join(temp1)
    temp2=temp2+'|'+temp[1]
    return temp2
variableArray=list("abcdefghijklmnopqrstuvwxyz")
variableArray2 = []
variableArray3=[]
```

```
variableArray5=[]
variableArray6=[]
for eachCombination in itertools.permutations(variableArray, 2):
   variableArray2.append(eachCombination[0]+eachCombination[1])
foreachCombinationinitertools.permutations(variableArray,3):
```

```
variableArray3.append(eachCombination[0]+eachCombination[1]+eachCombination[2]) for
eachCombination in itertools.permutations(variableArray, 4):
   variableArray5.append(eachCombination[0]+eachCombination[1]+eachCombination[2]+
eachCombination[3])
for eachCombination in itertools.permutations(variableArray, 5):
   variableArray6.append(eachCombination[0]+eachCombination[1]+eachCombination[2]+
eachCombination[3]+eachCombination[4])
variableArray=variableArray+variableArray2+variableArray3+variableArray5+variableArray6
capitalVariables = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
number=0

defstandardizationnew(sentence):
   newsentence=list(sentence)
   i=0
   global number
   variables=collections.OrderedDict()
   positionsofvariable=collections.OrderedDict()
   lengthofsentence=len(sentence)
   for i in range(0,lengthofsentence-1):
      if(newsentence[i]==','ornewsentence[i]=='('):
         if newsentence[i+1] not in capitalVariables:
            substitution=variables.get(newsentence[i+1])
            positionsofvariable[i+1]=i+1
            if not substitution :
               variables[newsentence[i+1]]=variableArray[number]
               newsentence[i+1]=variableArray[number]
               number+=1
            else:
               newsentence[i+1]=substitutionret
   urn"".join(newsentence)

def insidestandardizationnew(sentence):
   lengthofsentence=len(sentence)
   newsentence=sentence
   variables=collections.OrderedDict()
   positionsofvariable=collections.OrderedDict()
   global number
   i=0
   while i <=len(newsentence)-1 :
      if(newsentence[i]==','ornewsentence[i]=='('):
         ifnewsentence[i+1]notincapitalVariables:
```

```
            j=i+1
            while(newsentence[j]!=','andnewsentence[j]!=')'):
                j+=1
            substitution=variables.get(newsentence[i+1:j])
            if not substitution :
                variables[newsentence[i+1:j]]=variableArray[number]
                newsentence=newsentence[:i+1]+variableArray[number]+newsentence[j:]
                i=i+len(variableArray[number])
                number+=1
            else:
                newsentence=newsentence[:i+1]+substitution+newsentence[j:]
                i=i+len(substitution)
        i+=1
    returnnewsentence

def replace(sentence,theta):
    lengthofsentence=len(sentence)
    newsentence=sentence
    i=0
    while i <=len(newsentence)-1 :
        if(newsentence[i]==','ornewsentence[i]=='('):
            ifnewsentence[i+1]notincapitalVariables:
            j=i+1
            while(newsentence[j]!=','andnewsentence[j]!=')'):
                j+=1
            nstemp=newsentence[i+1:j]
            substitution=theta.get(nstemp)
            if substitution :
                newsentence=newsentence[:i+1]+substitution+newsentence[j:]
                i=i+len(substitution)
        i+=1
    return newsentence
repeatedsentencecheck=collections.OrderedDict()

def insidekbcheck(sentence):
    lengthofsentence=len(sentence)
    newsentence=pattern.split(sentence)
    newsentence.sort()
    newsentence="|".join(newsentence)
    global repeatedsentencecheck
    i=0
```

```
    while i <=len(newsentence)-1 :
        if(newsentence[i]==','ornewsentence[i]=='('):
            ifnewsentence[i+1]notincapitalVariables:
                j=i+1
                while(newsentence[j]!=','andnewsentence[j]!=')'):
                    j+=1
                newsentence=newsentence[:i+1]+'x'+newsentence[j:]
        i+=1
    repeatflag=repeatedsentencecheck.get(newsentence)
    if repeatflag :
        return True
    repeatedsentencecheck[newsentence]=1
    return False

for i in range(n+2,n+2+k):
     data1[i]=data1[i].replace("","")
     if "=>" in data1[i]:
        data1[i]=data1[i].replace("","")
        sentencetemp=CNF(data1[i].rstrip())
        kbbefore.append(sentencetemp)
     else:
        kbbefore.append(data1[i].rstrip())
for i in range(0,k):
    kbbefore[i]=kbbefore[i].replace("","")

kb={}
pattern=re.compile("\||&|=>")#wecanremovethe'\|"tospeedupas'OR'doesntcomeintheKB
pattern1=re.compile("[(,]")
for i in range(0,k):
    kbbefore[i]=standardizationnew(kbbefore[i])
    temp=pattern.split(kbbefore[i])
    lenoftemp=len(temp)
    forjinrange(0,lenoftemp):
        clause=temp[j]
        clause=clause[:-1]
        predicate=pattern1.split(clause)
        argumentlist=predicate[1:]
        lengthofpredicate=len(predicate)-1
        if predicate[0] in kb:
            if lengthofpredicate in kb[predicate[0]]:
                kb[predicate[0]][lengthofpredicate].append([kbbefore[i],temp,j,predicate[1:]])
```

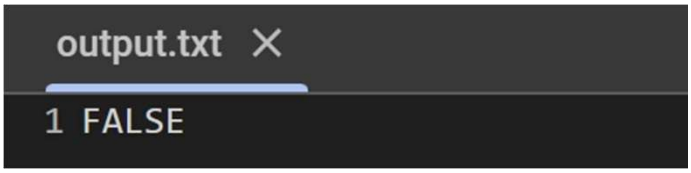
```
            else:
                kb[predicate[0]][lengthofpredicate]=[kbbefore[i],temp,j,predicate[1:]]
        else:
            kb[predicate[0]]={lengthofpredicate:[[kbbefore[i],temp,j,predicate[1:]]]}

for qi in range(0,n):
    queries[qi]=standardizationnew(queries[qi])

defsubstituevalue(paramArray,x,y):
    forindex,eachValinenumerate(paramArray): if
        eachVal == x:
            paramArray[index]=y
    return paramArray

def unificiation(arglist1,arglist2):
    theta=collections.OrderedDict()
    for i in range(len(arglist1)):
        ifarglist1[i]!=arglist2[i]and(arglist1[i][0]incapitalVariables)and(arglist2[i][0]in capitalVariables):
            return[]
        elifarglist1[i]==arglist2[i]and(arglist1[i][0]incapitalVariables)and(arglist2[i][0]in capitalVariables):
            ifarglist1[i]notintheta.keys():
                theta[arglist1[i]]=arglist2[i]
        elif(arglist1[i][0]incapitalVariables)andnot(arglist2[i][0]incapitalVariables): if
            arglist2[i] not in theta.keys():
                theta[arglist2[i]]=arglist1[i]
                arglist2=substituevalue(arglist2,arglist2[i],arglist1[i])
        elifnot(arglist1[i][0]incapitalVariables)and(arglist2[i][0]incapitalVariables): if
            arglist1[i] not in theta.keys():
                theta[arglist1[i]]=arglist2[i]
                arglist1=substituevalue(arglist1,arglist1[i],arglist2[i])
        elifnot(arglist1[i][0]incapitalVariables)andnot(arglist2[i][0]incapitalVariables): if
            arglist1[i] not in theta.keys():
                theta[arglist1[i]]=arglist2[i]
                arglist1=substituevalue(arglist1,arglist1[i],arglist2[i]) else:
                argval=theta[arglist1[i]]
                theta[arglist2[i]]=argval
                arglist2=substituevalue(arglist2,arglist2[i],argval)
    return [arglist1,arglist2,theta]
```

```
defresolution():
    globalrepeatedsentencecheck
    answer=list()
    qrno=0
    forqrinqueries:
        qrno+=1
        repeatedsentencecheck.clear()
        q=queue.Queue()
        query_start=time.time()
        kbquery=copy.deepcopy(kb)
        ans=qr
        ifqr[0]=='~':
            ans=qr[1:]
        else:
            ans='~'+qr
        q.put(ans)
        label:outerloop
        currentanswer="FALSE"
        counter=0
        while True:
            counter+=1
            ifq.empty():
                break
            ans=q.get()
            label:outerloop1
            ansclauses=pattern.split(ans)
            lenansclauses=len(ansclauses)
            flagmatchedwithkb=0
            innermostflag=0
            for ac in range(0,lenansclauses):
                insidekbflag=0
                ansclausestruncated=ansclauses[ac][:-1]
                ansclausespredicate=pattern1.split(ansclausestruncated)
                lenansclausespredicate=len(ansclausespredicate)-1
                if ansclausespredicate[0][0]=='~':
                    anspredicatenegated=ansclausespredicate[0][1:]
                else:
                    anspredicatenegated="~"+ansclausespredicate[0]
                x=kbquery.get(anspredicatenegated,{}).get(lenansclausespredicate)
                if not x:
```

```
            continue
        else:
            lenofx=len(x)
            fornumofpredinrange(0,lenofx):
                insidekbflag=0
                putinsideq=0
                sentenceselected=x[numofpred]

thetalist=unificiation(copy.deepcopy(sentenceselected[3]),copy.deepcopy(ansclausespredicate[1:]))
                if(len(thetalist)!=0):
                    for key in thetalist[2]:
                        tl=thetalist[2][key]
                        tl2=thetalist[2].get(tl)
                        if tl2:
                            thetalist[2][key]=tl2flagmatchedwithkb=1
                    notincludedindex=sentenceselected[2]
                    senclause=copy.deepcopy(sentenceselected[1])
                    mergepart1=""
                    del senclause[notincludedindex]
                    ansclauseleft=copy.deepcopy(ansclauses)
                    del ansclauseleft[ac]
                    for am in range(0,len(senclause)):
                        senclause[am]=replace(senclause[am],thetalist[2])
                        mergepart1=mergepart1+senclause[am]+'|'
                    for remain in range(0,len(ansclauseleft)):
                        listansclauseleft=ansclauseleft[remain]
                        ansclauseleft[remain]=replace(listansclauseleft,thetalist[2])
                        if ansclauseleft[remain] not in senclause:
                            mergepart1=mergepart1+ansclauseleft[remain]+'|'
                    mergepart1=mergepart1[:-1]
                    if mergepart1=="":
                      currentanswer="TRUE"
                      break
                    ckbflag=insidekbcheck(mergepart1)
                    if not ckbflag:
                            mergepart1=insidestandardizationnew(mergepart1)
                            ans=mergepart1
                            temp=pattern.split(ans)
                            lenoftemp=len(temp)
                            forjinrange(0,lenoftemp):
```

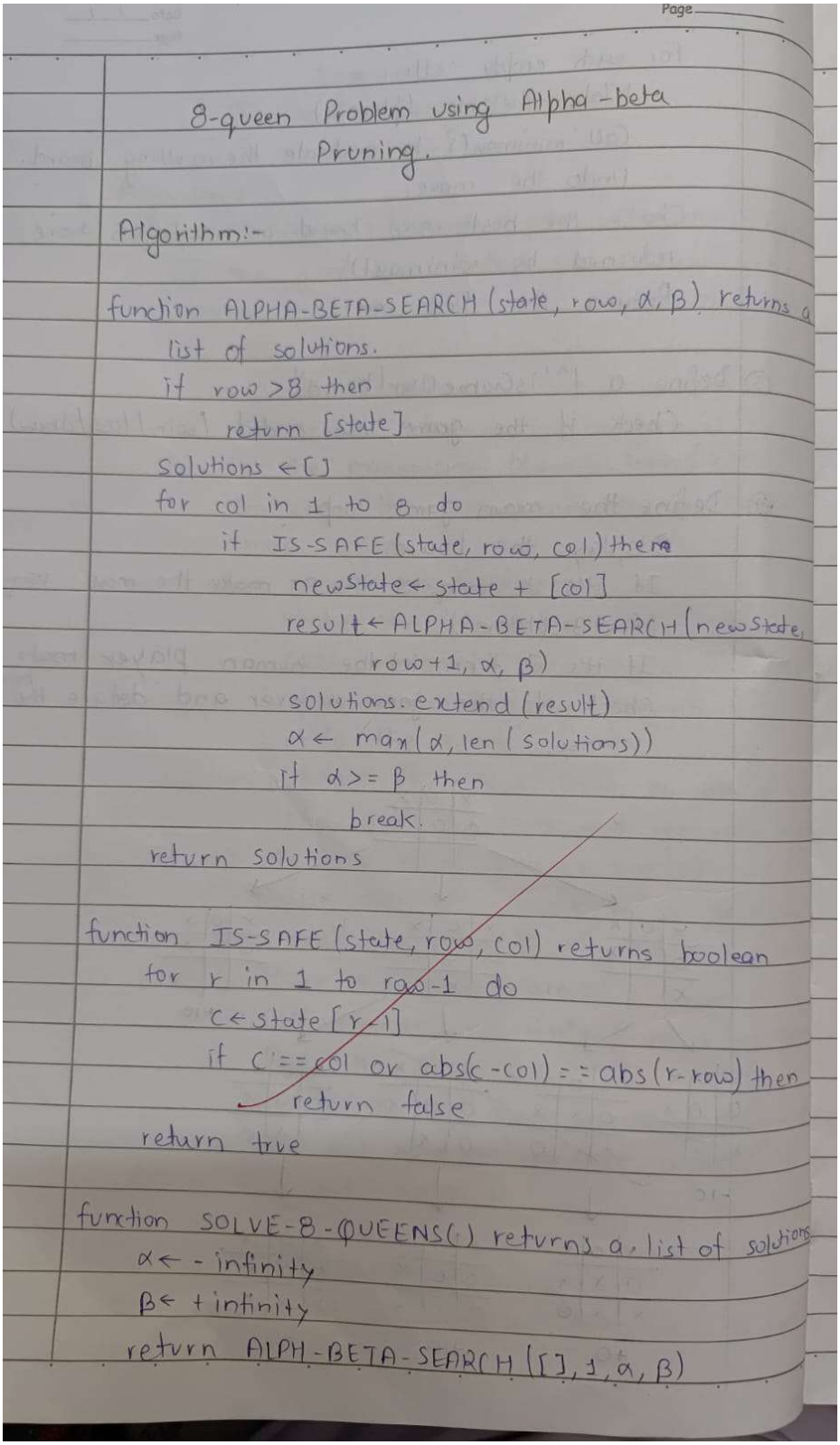
```
                clause=temp[j]
                clause=clause[:-1]
                predicate=pattern1.split(clause)
                argumentlist=predicate[1:]
                lengthofpredicate=len(predicate)-1
                if predicate[0] in kbquery:
                    iflengthofpredicateinkbquery[predicate[0]]:

kbquery[predicate[0]][lengthofpredicate].append([mergepart1,temp,j,argumentlist])
                        else:

kbquery[predicate[0]][lengthofpredicate]=[[mergepart1,temp,j,argumentlist]]
                else:

kbquery[predicate[0]]={lengthofpredicate:[[mergepart1,temp,j,argumentlist]]}
                    q.put(ans)
            if(currentanswer=="TRUE"):
                break
        if(currentanswer=="TRUE"):
            break
        if(counter==2000or(time.time()-query_start)>20):
            break
    answer.append(currentanswer)
  return answer

if name____== 'main':
  finalanswer=resolution()
  o=open("output.txt","w+")
  wc=0
  while(wc < n-1):
      o.write(finalanswer[wc]+"\n")
      wc+=1
  o.write(finalanswer[wc])
  o.close()
```

Output:



Alpha beta pruning :

Algorithm:



8-queen Problem using Alpha-beta Pruning.

Algorithm:-

```
function ALPHA-BETA-SEARCH (state, row, α, β) returns a
    list of solutions.
    if row > 8 then
        return [state]
    solutions ← []
    for col in 1 to 8 do
        if IS-SAFE (state, row, col) then
            newstate ← state + [col]
            result ← ALPHA-BETA-SEARCH (newstate,
                row+1, α, β)
            solutions.extend (result)
            α ← max (α, len (solutions))
            if α >= β then
                break.
    return solutions

function IS-SAFE (state, row, col) returns boolean
    for r in 1 to row-1 do
        c ← state [r-1]
        if c == col or abs(c-col) == abs(r-row) then
            return false
    return true

function SOLVE-8-QUEENS() returns a list of solutions
    α ← - infinity
    β ← + infinity
    return ALPH-BETA-SEARCH ([], 1, α, β)
```

Code:

```
import math

def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta):
    # Base case: If the depth is 0, return the value at this node
    if depth == 0:
        return values[node_index]

    if maximizing_player:
        max_eval = -math.inf  # Start with a very small value
        for i in range(2):  # Assuming binary tree structure with two children
            eval_value = alpha_beta(depth - 1, node_index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval_value)
            alpha = max(alpha, eval_value)
            if beta <= alpha:
                break  # Beta cut-off
        return max_eval
    else:
        min_eval = math.inf  # Start with a very large value
        for i in range(2):  # Assuming binary tree structure with two children
            eval_value = alpha_beta(depth - 1, node_index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval_value)
            beta = min(beta, eval_value)
            if beta <= alpha:
                break  # Alpha cut-off
        return min_eval

def main():
    depth = int(input("Enter the depth of the tree: "))
    num_nodes = 2 ** (depth + 1) - 1  # Number of nodes in a binary tree
    values = []

    print(f"Enter the leaf node values for depth {depth}:")
    for i in range(2 ** depth):  # Leaf nodes are at the last level
        value = int(input(f"Enter value for leaf node {i + 1}: "))
        values.append(value)

    alpha = -math.inf
    beta = math.inf
    result = alpha_beta(depth, 0, True, values, alpha, beta)

    print(f"Optimal value (using Alpha-Beta Pruning): {result}")

if __name__ == "__main__":
    main()
```

Output:

```
Enter the depth of the tree: 3
Enter the leaf node values for depth 3:
Enter value for leaf node 1: 10
Enter value for leaf node 2: 9
Enter value for leaf node 3: 14
Enter value for leaf node 4: 18
Enter value for leaf node 5: 5
Enter value for leaf node 6: 4
Enter value for leaf node 7: 50
Enter value for leaf node 8: 3
Optimal value (using Alpha-Beta Pruning): 10
```