

**VISVESVARAYA TECHNOLOGICAL
UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Anshuman Gupta (1BM23CS043)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Aug 2025 to Dec 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 out by **Ansuman Gupta(1BM23CS043)**, 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.

Dr. K.R. Mamatha Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
---	--

Index

Sl. No.	Date	Experiment Title	Page No.
1	18-08-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4-9
2	25-08-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	10-19
3	25-08-2025	Implement A* search algorithm	20-25
4	15-09-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	26-29
5	15-09-2025	Simulated Annealing to Solve 8-Queens problem	30-32
6	22-09-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	33-37
7	13-10-2025	Implement unification in first order logic	38-43
8	13-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	44-47
9	27-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	48-51
10	16-11-2025	Implement Alpha-Beta Pruning.	52-55

Github Link: https://github.com/Anshuman262005/AI_LAB



CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Anshuman Anshuman

for successfully completing

Artificial Intelligence Foundation Certification

on November 25, 2025



Infosys | Springboard

Congratulations! You make us proud!

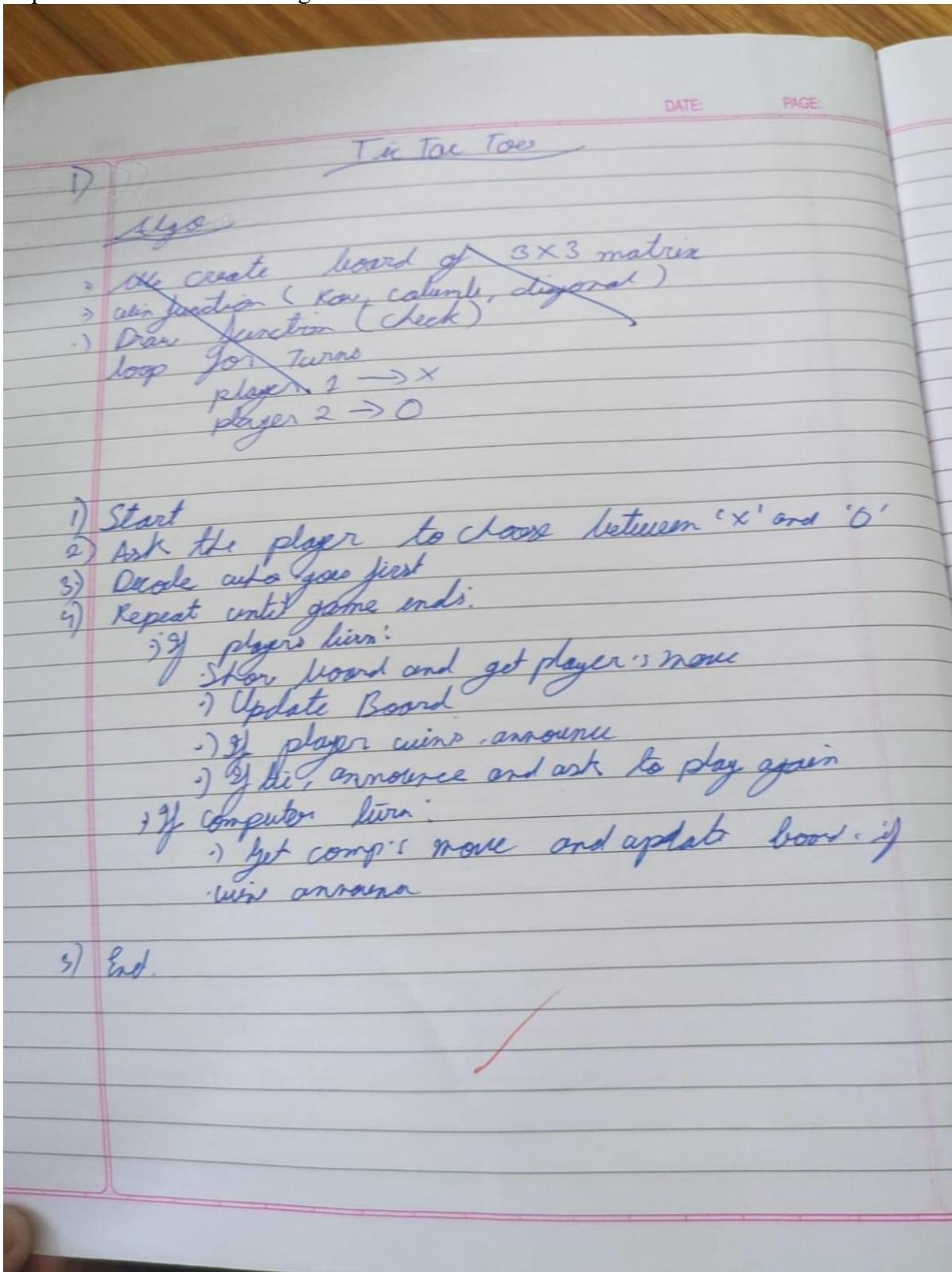
Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at <https://verify.onwingspan.com>

Satheesha B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Program 1

Implement Tic - Tac - Toe Game

Implement vacuum cleaner agent



Pseudocode

DATE:

PAGE:

Create board of 3×3 matrix

def print_board (board): display the board
in 3×3 grid

function minimax (board, is_max):

if AI win - return +1

if human wins - return -1

if Draw → Return 0

if is MAX:

best = -∞

for each empty cell:

simulate AI move

best = MAX (best, minimax (board,
false))

Return best.

else:

best = +∞

for each empty cell:

simulate Human Move

best = min (best, minimax (board, true))

Return best.

Function find_best_move (board):

best score = -∞

for each empty cell:

simulate AI move

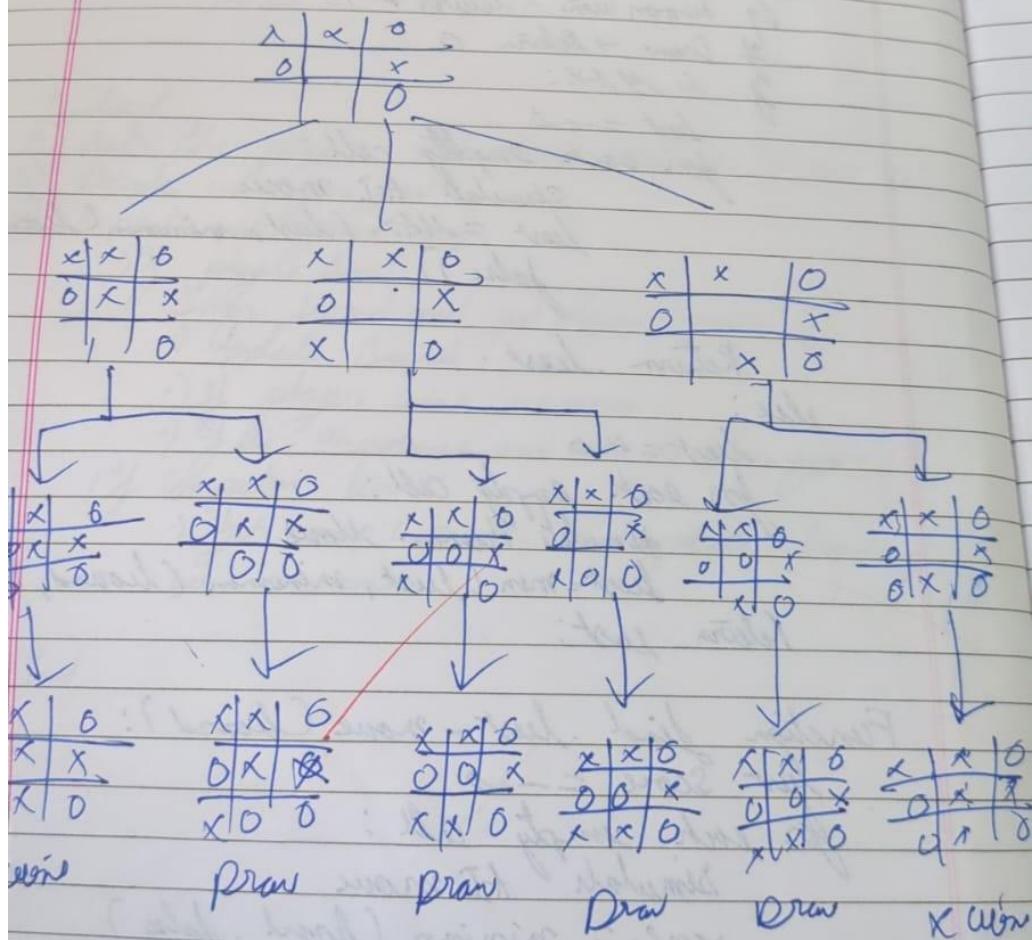
score = minimax (board, false)?

end loop

If score > best score
 best score = score
 best move = all.

Output

Initial state : X



Vacuum Cleaner

Alg:

- 1) Divide environment into 4 quadrants ($Q_1 - Q_4$)
- 2) Assign each quadrant randomly as clean/dirty.
- 3) Start from Q_1
- 4) at each step :
 - a) if dirt \rightarrow clean
 - b) else \rightarrow skip
- 5) Move clockwise to next quadrant
- 6) Repeat for required cycles.
- 7) print final status.

Det Pat

Action : one of {L, R, T, D, U, S}

which clean at least one Dirty cell
in grid.

$x, y \leftarrow$ position
 check grid (x, y) is Dirty then
 perform cleaning
 If not
 if there is an dirty cell adjacent.
 Step choose a direction (L, R, U, D)
 toward that cell.
 Move to the new position.

else
 move here by row to cover all cell.
 stop out put.

$\begin{bmatrix} v \\ p \end{bmatrix} \rightarrow \begin{bmatrix} p \\ v \end{bmatrix} \rightarrow \begin{bmatrix} v \\ p \end{bmatrix} \rightarrow \begin{bmatrix} v \\ v \end{bmatrix}$

Kom is clean!

Baudouin
class VacuumCleaner:

INT T:

quadrant = $[0_1, 0_2, 0_3, 0_4]$

slotw = random (clean / dirty) for each quadrant

current - ender = 0

Function clean (q)
mark q as clean

Function move - next v:
move clockwise to next quadrant.

Function start (cycles)

Repeat cycles x 4:

current = quadrant (current - ender)

if slotw (current) == dirty :

 clean (current)

else :

 skip

 move - next ()

Print final status.

Code:

Tic tac toe:

```
def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):
            return True
        if all(board[j][i] == player for j in range(3)):
            return True
        if all(board[i][i] == player for i in range(3)):
            return True
        if all(board[i][2 - i] == player for i in range(3)):
            return True
    return False

def is_draw(board):
    return all(board[i][j] != '-' for i in range(3) for j in range(3))

def minimax(board, is_ai_turn):
    if check_winner(board, 'O'):
        return 1
    if check_winner(board, 'X'):
        return -1
    if is_draw(board):
        return 0

    if is_ai_turn:
        best_score = -float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == '-':
                    board[i][j] = 'O'
                    score = minimax(board, False)
                    board[i][j] = '-'
                    best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == '-':
                    board[i][j] = 'X'
                    score = minimax(board, True)
                    board[i][j] = '-'
                    best_score = min(score, best_score)
        return best_score
```

```

        board[i][j] = '-'
        best_score = min(score, best_score)
    return best_score

def manual_game():
    board = [[ '-' for _ in range(3)] for _ in range(3)]
    print("Initial Board:")
    print_board(board)

    while True:
        while True:
            try:
                x_row = int(input("Enter X row (1-3): ")) - 1
                x_col = int(input("Enter X col (1-3): ")) - 1
                if board[x_row][x_col] == '-':
                    board[x_row][x_col] = 'X'
                    break
                else:
                    print("Cell occupied!")
            except:
                print("Invalid input!")

    print("Board after X move:")
    print_board(board)

    if check_winner(board, 'X'):
        print("X wins!")
        break
    if is_draw(board):
        print("Draw!")
        break

    while True:
        try:
            o_row = int(input("Enter O row (1-3): ")) - 1
            o_col = int(input("Enter O col (1-3): ")) - 1
            if board[o_row][o_col] == '-':
                board[o_row][o_col] = 'O'
                break
            else:
                print("Cell occupied!")
        except:
            print("Invalid input!")

    print("Board after O move:")
    print_board(board)

```

```

        if check_winner(board, 'O'):
            print("O wins!")
            break
        if is_draw(board):
            print("Draw!")
            break

        cost = minimax(board, True)
        print(f"AI evaluation cost from this position: {cost}")

    manual_game()
    print("Name: Sanchit Mehta and USN : 1BM23CS299")

vacuum cleaner:
rooms = int(input("Enter Number of rooms: "))
Rooms = "ABCDEFGHIJKLMNPQRSTUVWXYZ"
cost = 0
Roomval = {}

for i in range(rooms):
    print(f"Enter Room {Rooms[i]} state (0 for clean, 1 for dirty): ")
    n = int(input())
    Roomval[Rooms[i]] = n

loc = input(f"Enter Location of vacuum ({Rooms[:rooms]}): ").upper()

while 1 in Roomval.values():
    if Roomval[loc] == 1:
        print(f"Room {loc} is dirty. Cleaning...")
        Roomval[loc] = 0
        cost += 1
    else:
        print(f"Room {loc} is already clean.")

move = input("Enter L or R to move left or right (or Q to quit): ").upper()

if move == "L":
    if loc != Rooms[0]:
        loc = Rooms[Rooms.index(loc) - 1]
    else:
        print("No room to move left.")
elif move == "R":
    if loc != Rooms[rooms - 1]:
        loc = Rooms[Rooms.index(loc) + 1]
    else:
        print("No room to move right.")

```

```
elif move == "Q":  
    break  
else:  
    print("Invalid input. Please enter L, R, or Q.")  
  
print("\nAll Rooms Cleaned." if l not in Roomval.values() else "Exited before cleaning all rooms.")  
print(f"Total cost: {cost}")
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS):

8 PU22 LE (NeuralNet)

DATE: _____

PAGE: _____

Alpha Beta Pruning

Start with initial state.

FUNCTION $\text{AStar}(\text{start_state}, \text{goal_state})$:

DEFINE priority queue PA

INSERT $(f=0, g=0, \text{state} = \text{start_state}, \text{path} = [\text{start_state}])$ INTO PA .

DEFINE visited set = \emptyset

WHILE PA is not empty do

$(f, g, \text{state}, \text{path}) \leftarrow$ remove smallest f from PA

IF state is in visited set:

Continue

ADD state TO visited set:

if $\text{state} == \text{goal_state}$:

return $\text{path} + [\text{state}]$

For each neighbour of state

If neighbour not in visited set:

$h = \text{heuristic(neighbour)}$

$f = g + f + h$

Insert $(f, g, h, \text{neighbour}, \text{path} + [\text{state}])$

Function `neighbours (slab)`:

count = 0
for each tile at position (i, j):
if tile $i = 0$ & tile $j = \text{goal slab}$ (C, D)
count = count + 1
return count

Function `get_neighbours (slab)`:

Find blank - position (x, y)

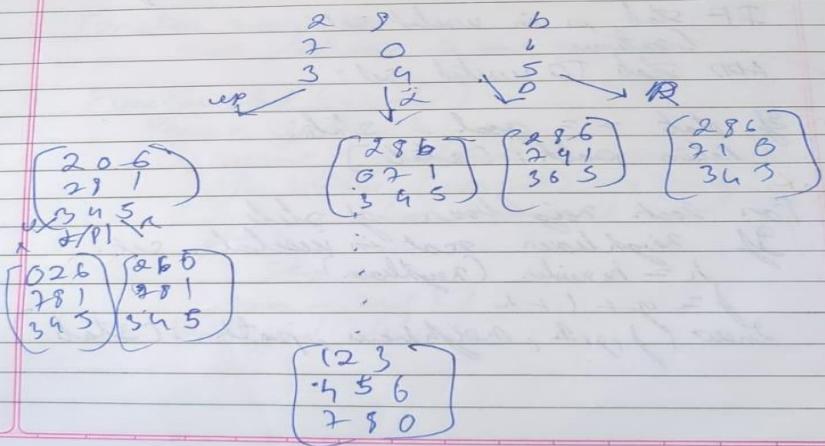
Define moves - (up), ($down$), ($left$), ($right$)

Four lab valid move:

swap blank with adjacent tiles

Add new slab to neighbour.

return neighbours



Perform IDDFS

procedure IDDFS (root, goal-test):

depth ← 0

loop forever:

found, path ← DLS (root, depth, goal-test,
stack-path = {root})

if found = True:

return path
depth ← depth + 1

procedure DLS (node, limit, goal-test, stack):

if goal-test (node) = True:

return (True, stack-path)

if limit = 0:

return (False, None)

for each-child in expand (node)

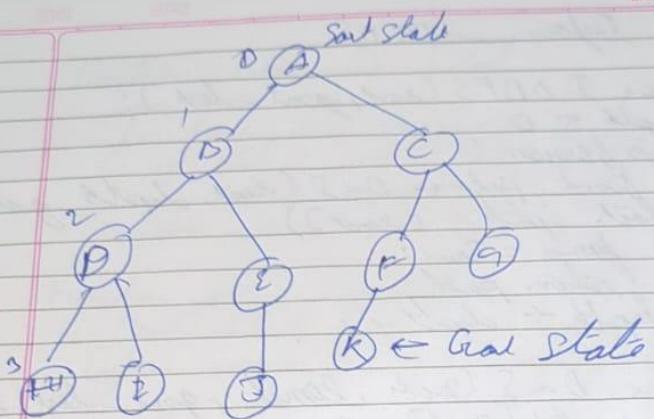
if child not in stack-path

found, path ← DLS (child, limit - 1,
goal-test, stack-path + {child})

if found = True

return (True, path)

return (False, None).



Depth = 0

visit = A

goal not found

Depth = 1

visit = A, D, C

goal not found

Depth = 2

visit = A, B, D, E, C, F, G

goal not found.

Depth = 3

visit = A, B, D, H, I, E, S, C, F, R, G

goal found.

The goal state is found at depth 3

DATE: PAGE:

8 puzzle (misplaced)

1) Start with initial state of the puzzle and set depth limit = 0

Misplace Tiles heuristic

Algo:

Count = 0

2) for each tile in puzzle (except blank):
if tile is not in its correct position.
→ increment count

3) return count

Pseudocode

function misplaced-Tiles (state, goal)

count ← 0

for i from 0 to 8

if state [i] ≠ -1 and
state [i] ≠ goal [i]

count ← count + 1

return count.

Example

5	2	1
7	3	6
2	3	6

Znike's state

1	2	3
4	5	6
7	8	

Goal state

misplaced = 6 tiles

```

goal_state = '123804765'

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}
count = 0
invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'], 5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

def print_state(state):
    for i in range(0, 9, 3):
        print(''.join(state[i:i+3]).replace('0', ' '))
    print()

def dfs(start_state, max_depth=50):
    visited = set()
    stack = [(start_state, [])] # Each element: (state, path)

    while stack:
        current_state, path = stack.pop()

        if current_state in visited:
            continue

        # Print every visited state
        print("Visited state:")
        print_state(current_state)

        if current_state == goal_state:
            return path

        for direction in moves:
            new_index = path[-1] + moves[direction]
            if new_index < 0 or new_index >= 9:
                continue

            new_state = list(current_state)
            new_state[path[-1]] = '0'
            new_state[new_index] = direction
            new_path = path + [new_index]

            if new_state not in visited:
                stack.append((new_state, new_path))
                visited.add(new_state)

```

```

visited.add(current_state)
global count
count +=1
if len(path) >= max_depth:
    continue

for direction in moves:
    new_state = move_tile(current_state, direction)
    if new_state and new_state not in visited:
        stack.append((new_state, path + [direction]))

return None

start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)

    result = dfs(start)

    if result is not None:
        print("Solution found!")
        print("Moves:", ''.join(result))
        print("Number of moves:", len(result))
        print("Number of visited states", count)

        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f"Move {i}: {move}")
            print_state(current_state)
    else:
        print("No solution exists for the given start state or max depth reached.")

else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")

```

Implement Iterative deepening search algorithm:
goal_state = '123456780'

```

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'],
    1: ['U'],
    2: ['U', 'R'],
    3: ['L'],
    5: ['R'],
}

```

```

6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

def print_state(state):
    for i in range(0, 9, 3):
        print(''.join(state[i:i+3]).replace('0', ' '))
    print()

def dls(state, depth, path, visited, visited_count):
    visited_count[0] += 1 # Increment visited states count
    if state == goal_state:
        return path

    if depth == 0:
        return None

    visited.add(state)

    for direction in moves:
        new_state = move_tile(state, direction)
        if new_state and new_state not in visited:
            result = dls(new_state, depth - 1, path + [direction], visited, visited_count)
            if result is not None:
                return result

    visited.remove(state)
    return None

def iddfs(start_state, max_depth=50):
    visited_count = [0] # Using list to pass by reference
    for depth in range(max_depth + 1):
        visited = set()
        result = dls(start_state, depth, [], visited, visited_count)
        if result is not None:

```

```

        return result, visited_count[0]
    return None, visited_count[0]

# Main
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)

    result, visited_states = iddfs(start,15)

    print(f'Total states visited: {visited_states}')

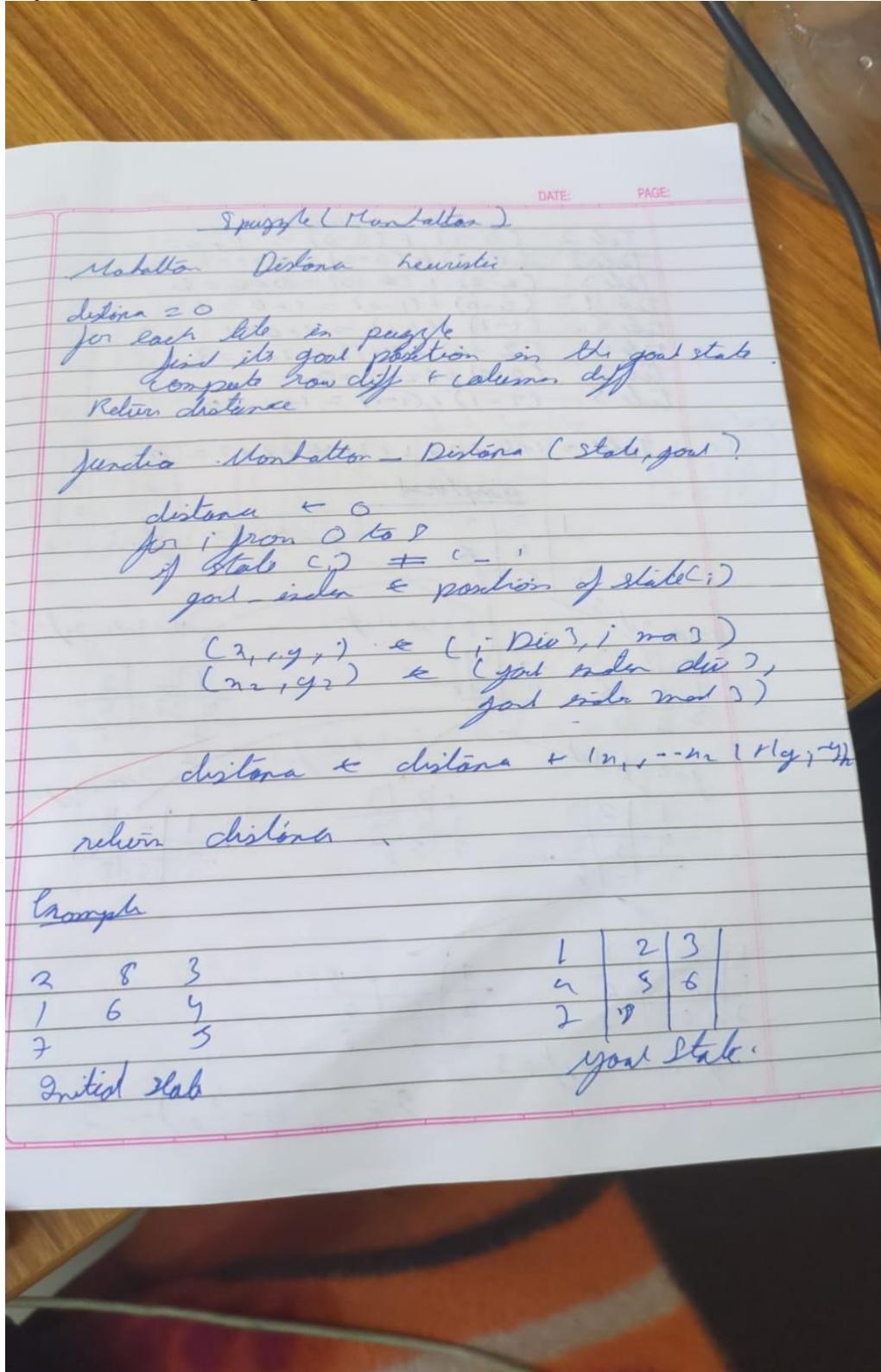
    if result is not None:
        print("Solution found!")
        print("Moves:", ''.join(result))
        print("Number of moves:", len(result))

        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f'Move {i}: {move}')
            print_state(current_state)
    else:
        print("No solution exists for the given start state or max depth reached.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")

```

Program 3

Implement A* search algorithm



- Tab 2: $(0-1) + (0-0) = 1+0=1$
 Tab 3: $(1-1) + (0+2) = 0+2=2$
 Tab 5: $(0-0) \cup (0-1) = 0+0=0$
 Tab 1: $(0-0) + (1-0) = 1+0=1$
 Tab 6: $(1-2) \cup (1-1) = 1+0=1$
 Tab 9: $(2-0) + (1-1) = 2+0=2$
 Tab 7: $(0-0) \cup (2-2) = 0$
 Tab 5: $(2-1) \cup (2-1) = 1+1=2$

~~$\text{Total} = 1+2+0+1+1+2+2+2 = 9$~~

misplaced

1	2	3
4	6	
7	5	8

$y=0, h=3, f=3$

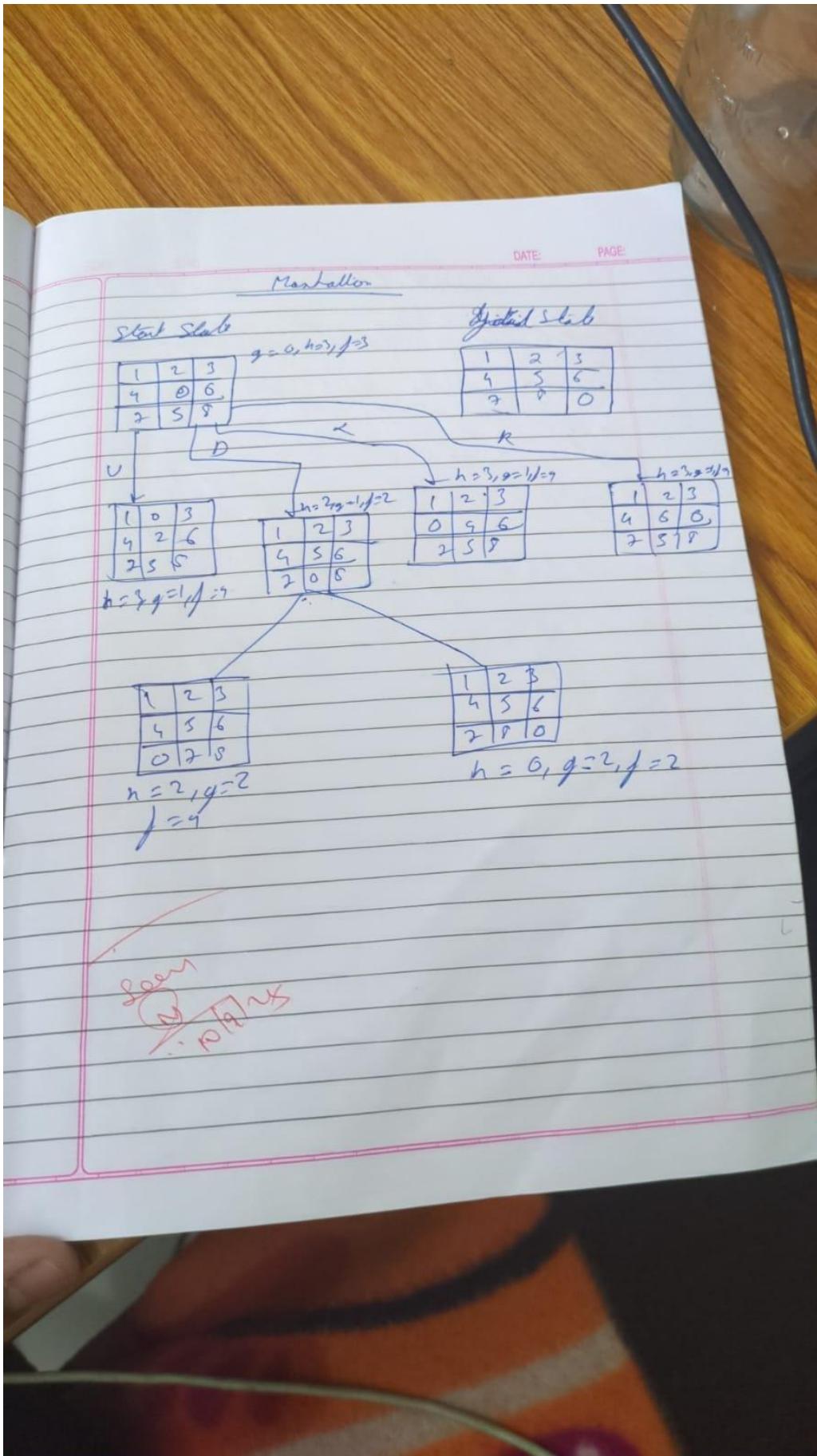
$y=1, h=1, f=5$	$y=1, h=2, f=3$	$y=1, h=3, f=5$
$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$	$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$	$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$
$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$	$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$	$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$
$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$	$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$	$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$
$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$	$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$	$\begin{array}{ c c }\hline 1 & 2 \\ \hline 4 & 6 \\ \hline 7 & 5 \\ \hline \end{array}$

1	2	3
4	5	6
7	8	-

$y=3, h=0, f=3$

9	2	3
4	5	6
7	2	8

$y=3, h=2, f=3$



```

import heapq

goal_state = '123456780'

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'],
    1: ['U'],
    2: ['U', 'R'],
    3: ['L'],
    5: ['R'],
    6: ['D', 'L'],
    7: ['D'],
    8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

def print_state(state):
    for i in range(0, 9, 3):
        print(''.join(state[i:i+3]).replace('0', ' '))
    print()

def manhattan_distance(state):
    distance = 0
    for i, val in enumerate(state):
        if val == '0':
            continue
        goal_pos = int(val) - 1
        current_row, current_col = divmod(i, 3)
        goal_row, goal_col = divmod(goal_pos, 3)
        distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

```

```

def a_star(start_state):
    visited_count = 0
    open_set = []
    heapq.heappush(open_set, (manhattan_distance(start_state), 0, start_state, []))
    visited = set()

    while open_set:
        f, g, current_state, path = heapq.heappop(open_set)
        visited_count += 1

        if current_state == goal_state:
            return path, visited_count

        if current_state in visited:
            continue
        visited.add(current_state)

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(new_state)
                heapq.heappush(open_set, (new_f, new_g, new_state, path + [direction]))

    return None, visited_count

# Main
start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)

    result, visited_states = a_star(start)

    print(f"Total states visited: {visited_states}")

    if result is not None:
        print("Solution found!")
        print("Moves:", ''.join(result))
        print("Number of moves:", len(result))

        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f"Move {i}: {move}")

```

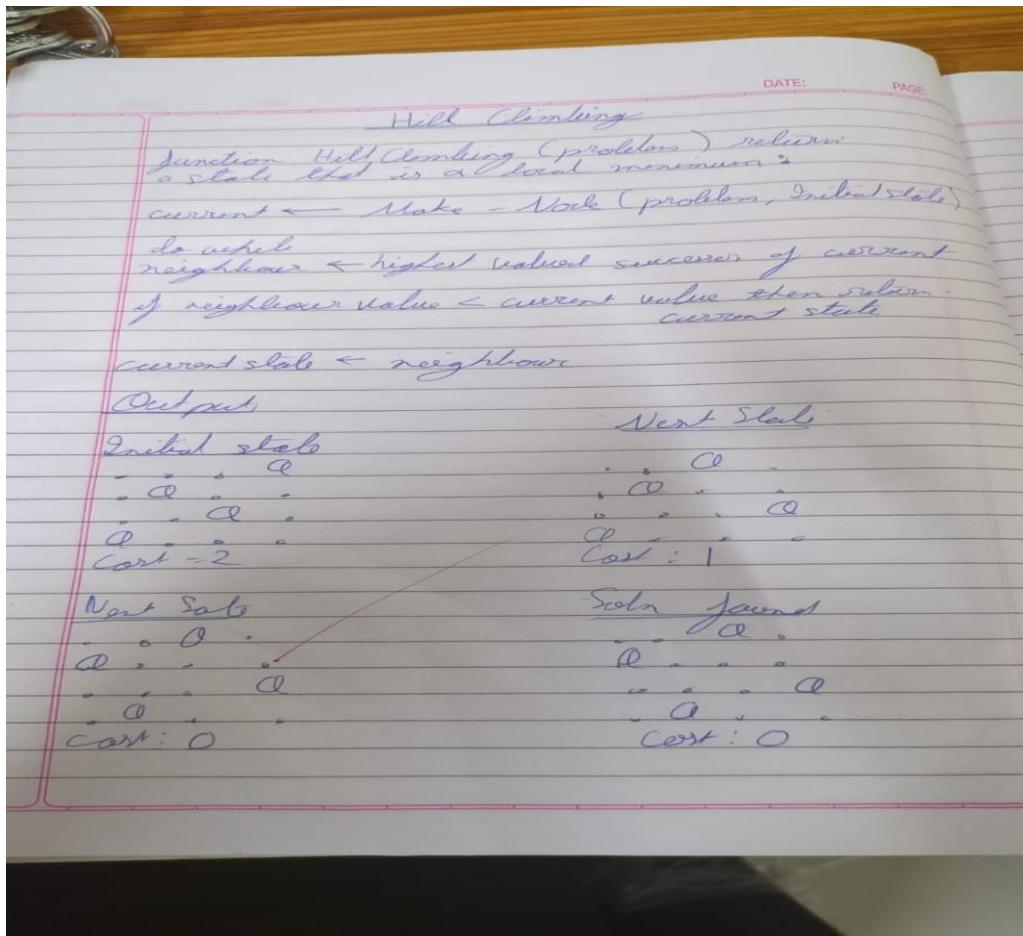
```

        print_state(current_state)
else:
    print("No solution exists for the given start state.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")

```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem



```

import random
import time

```

```

def print_board(state):
    """Prints the chessboard for a given state."""

```

```

n = len(state)
for row in range(n):
    line = ""
    for col in range(n):
        if state[col] == row:
            line += "Q "
        else:
            line += "."
    print(line)
print()

def compute_heuristic(state):
    """Computes the number of attacking pairs of queens."""
    h = 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):
                h += 1
    return h

def get_neighbors(state):
    """Generates all possible neighbors by moving one queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if row != state[col]:
                neighbor = list(state)
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors

def hill_climb_verbose(initial_state, step_delay=0.5):
    """Hill climbing algorithm with verbose output at each step."""
    current = initial_state
    current_h = compute_heuristic(current)
    step = 0

    print(f'Initial state (heuristic: {current_h}):')
    print_board(current)
    time.sleep(step_delay)

    while True:
        neighbors = get_neighbors(current)
        next_state = None
        next_h = current_h

```

```

for neighbor in neighbors:
    h = compute_heuristic(neighbor)
    if h < next_h:
        next_state = neighbor
        next_h = h

if next_h >= current_h:
    print(f'Reached local minimum at step {step}, heuristic: {current_h}')
    return current, current_h

current = next_state
current_h = next_h
step += 1
print(f'Step {step}: (heuristic: {current_h})')
print_board(current)
time.sleep(step_delay)

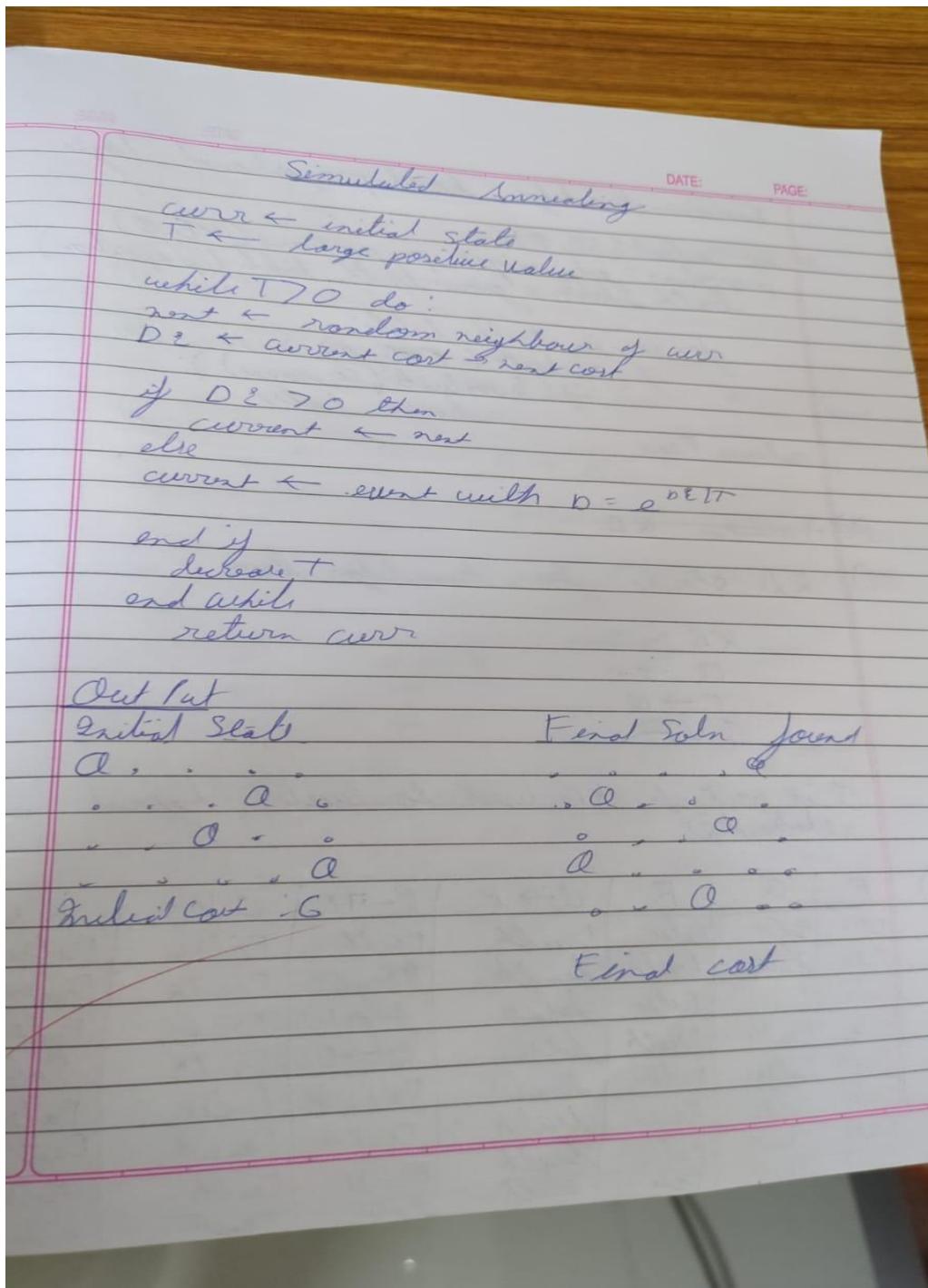
def solve_n_queens_verbose(n, max_restarts=1000):
    """Solves N-Queens problem using hill climbing with restarts and verbose output."""
    for attempt in range(max_restarts):
        print(f'\n==== Restart {attempt + 1} ====\n')
        initial_state = [random.randint(0, n - 1) for _ in range(n)]
        solution, h = hill_climb_verbose(initial_state)
        if h == 0:
            print(f' Solution found after {attempt + 1} restart(s):')
            print_board(solution)
            return solution
        else:
            print(f'No solution in this attempt (local minimum).\n')
    print("Failed to find a solution after max restarts.")
    return None

# --- Run the algorithm ---
if __name__ == "__main__":
    N = int(input("Enter the number of queens (N): "))
    solve_n_queens_verbose(N)

```

Program 5

Simulated Annealing to Solve 8-Queens problem



```

import random
import math

def compute_heuristic(state):
    """Number of attacking pairs."""
    h = 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):
                h += 1
    return h

def random_neighborhood(state):
    """Returns a neighbor by randomly changing one queen's row."""
    n = len(state)
    neighbor = state[:]
    col = random.randint(0, n - 1)
    old_row = neighbor[col]
    new_row = random.choice([r for r in range(n) if r != old_row])
    neighbor[col] = new_row
    return neighbor

def dual_simulated_annealing(n, max_iter=10000, initial_temp=100.0, cooling_rate=0.99):
    """Simulated Annealing with dual acceptance strategy."""
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_h = compute_heuristic(current)
    temperature = initial_temp

    for step in range(max_iter):
        if current_h == 0:
            print(f" ✅ Solution found at step {step}")
            return current

        neighbor = random_neighborhood(current)
        neighbor_h = compute_heuristic(neighbor)
        delta = neighbor_h - current_h

        if delta < 0:
            current = neighbor
            current_h = neighbor_h
        else:
            # Dual acceptance: standard + small chance of higher uphill move
            probability = math.exp(-delta / temperature)

```

```

if random.random() < probability:
    current = neighbor
    current_h = neighbor_h

temperature *= cooling_rate
if temperature < 1e-5: # Restart if stuck
    temperature = initial_temp
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_h = compute_heuristic(current)

print("✖ Failed to find solution within max iterations.")
return None

# --- Run the algorithm ---
if __name__ == "__main__":
    N = int(input("Enter number of queens (N): "))
    solution = dual_simulated_annealing(N)

if solution:
    print("Position format:")
    print("[", " ".join(str(x) for x in solution), "]")
    print("Heuristic:", compute_heuristic(solution))

```

Program 6

Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not:

Knowledge Knowledge based propositional logic							
							Date: _____
							Page: _____
# def entails (KB, Q):							
	propositions = get unique propositions (KB, Q)						
	truth table = gen built tabl (proposition)						
for row in truth table:							
	if evaluable (KB, Q):						
	if (! evaluable (Q, row)): return False.						
	return True						
Q1) what is KB →							
Q2) KB stands for knowledge base							
	KB contains						
	$\alpha \rightarrow \beta$						
	$\beta \rightarrow \alpha$						
	$\alpha \vee \beta$						
α is part of KB used to build logical relationship.							
i)	P	α	R	$\alpha \rightarrow P$	$P \rightarrow Q$	$\alpha \vee R$	KB
	False	False	False	truth	truth	False	False
	False	False	truth	truth	False	truth	truth
	False	true	False	false	False	False	False
	False	truth	truth	false	False	truth	False
	truth	False	False	truth	True	False	False
	truth	False	true	truth	True	truth	truth
	truth	truth	False	truth	False	truth	False
	truth	truth	True	truth	True	truth	True

i)

P	Q	R
F	F	T
+ F		T

DATE

PAGE

KB entails R

You are the two models where K is true

ii)

KB entails R

Since antecedent KB is true, K is true

iii)

P	K	$R \rightarrow P$
F	T	F
F	T	T

So, KB does not entail $R \rightarrow P$

iv)

Q	K	$Q \rightarrow K$
F	F	T
F	T	T

So, KB does not entail $Q \rightarrow K$

proceed

⊗: 1st order

```

from typing import List, Tuple, Dict, Set, Union

Predicate = Tuple[str, Tuple[str, ...]]

class Rule:
    def __init__(self, head: Predicate, body: List[Predicate]):
        self.head = head
        self.body = body

    def __repr__(self):
        body_str = ', '.join(f'{p[0]} {p[1]}' for p in self.body)
        return f'{body_str} => {self.head[0]} {self.head[1]}'

# Knowledge base
class KnowledgeBase:
    def __init__(self):
        self.facts: Set[Predicate] = set()
        self.rules: List[Rule] = []

    def add_fact(self, fact: Predicate):
        self.facts.add(fact)

    def add_rule(self, rule: Rule):
        self.rules.append(rule)

    def forward_chain(self, query: Predicate) -> bool:
        inferred = set(self.facts)
        added = True

        while added:
            added = False
            for rule in self.rules:
                if all(self._match_fact(body_pred, inferred) for body_pred in rule.body):
                    if not self._match_fact(rule.head, inferred):
                        inferred.add(rule.head)
                        added = True
                        print(f'Inferred: {rule.head}')

                if self._match_fact(query, inferred):
                    return True
            return self._match_fact(query, inferred)

    def _match_fact(self, pred: Predicate, fact_set: Set[Predicate]) -> bool:
        return pred in fact_set

```

```

# --- Example usage ---
if __name__ == "__main__":
    kb = KnowledgeBase()

    kb.add_fact(("Parent", ("John", "Mary")))
    kb.add_fact(("Parent", ("Mary", "Sue")))

    facts_list = list(kb.facts)
    for f1 in facts_list:
        for f2 in facts_list:
            if f1[0] == "Parent" and f2[0] == "Parent":
                if f1[1][1] == f2[1][0]:
                    head = ("Grandparent", (f1[1][0], f2[1][1]))
                    body = [f1, f2]
                    kb.add_rule(Rule(head, body))

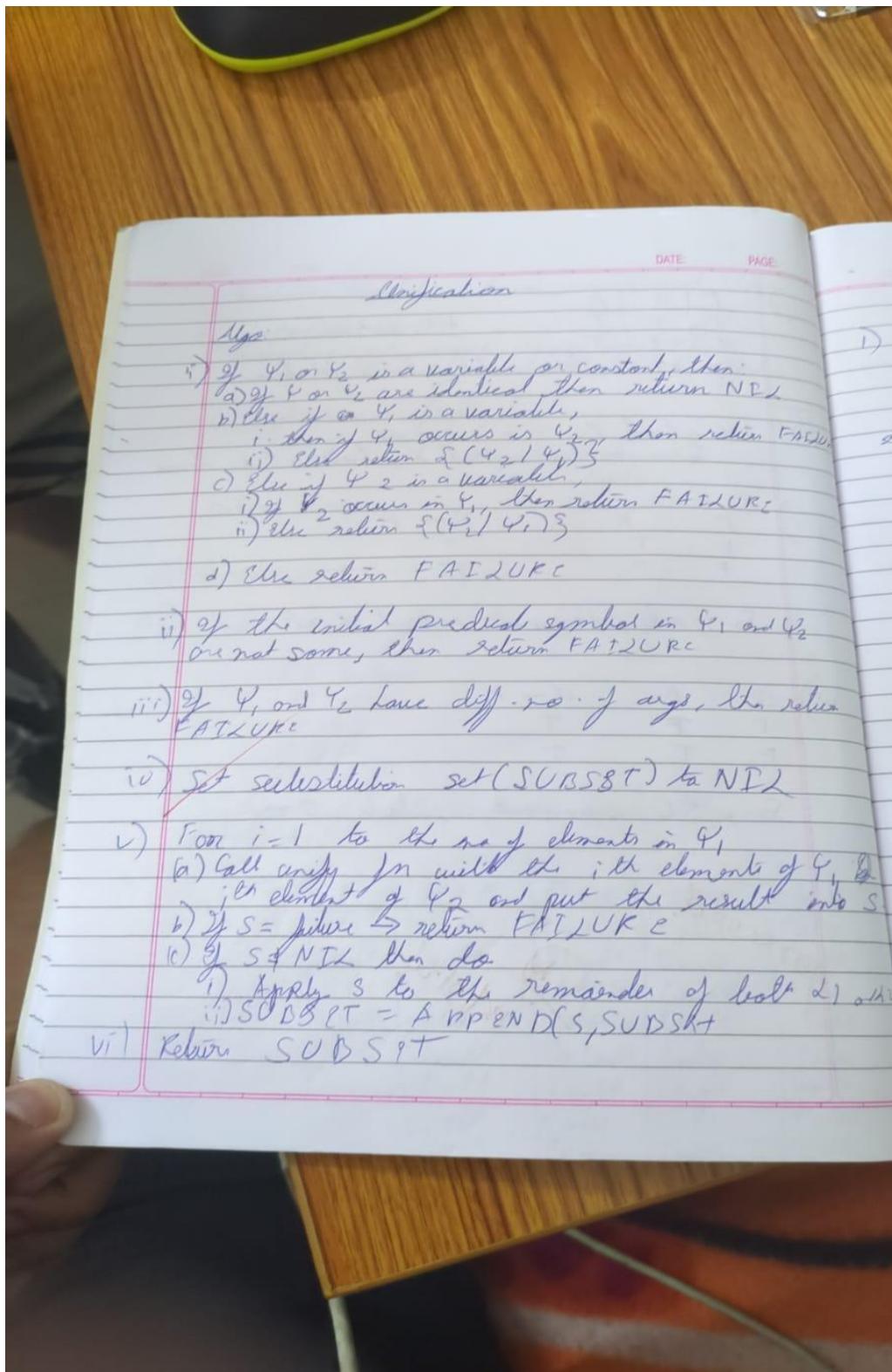
    query = ("Grandparent", ("John", "Sue"))
    result = kb.forward_chain(query)

    print(f"Query {query} is", "True" if result else "False")

```

Program 7

Implement unification in first order logic:



PAGE: _____ DATE: _____ PAGE: _____

Ex:

1) Unify ($\text{k}(\text{John}, n), \text{k}(\text{Sara}, \text{Jane})$)
 $\theta = \{n \rightarrow \text{Jane}\}$
 $\text{Unify}(\text{k}(\text{John}, \text{Jane}), \text{k}(\text{Sara}, \text{Jane}))$

2) Unify ($\text{k}(\text{John}, n), \text{k}(\text{Sara}, \text{B}'\text{II}))$
 $\theta = \{n \rightarrow \text{B}'\text{II}\}$
 $\text{Unify}(\text{k}(\text{John}, \text{B}'\text{II}), \text{k}(\text{Sara}, \text{B}'\text{II}))$

3) Check for $\{p(b, n, f(y\text{II})), p(z, f(i), f(g(z)))\}$ and $p(a, g(f(b), a), z)$
 $\{z \rightarrow b, n \rightarrow f(:), i \rightarrow g(z)\}$

ii) $\{p(g(g(a, a), f(g)), p(a, g(f(b), a), z))$
 $\{a \rightarrow b, g \rightarrow f\}$

iii) $\{p(f(a), f(g(x)), f(x, y))\}$
 $x = f(a)$ and $x = f(y)$
 $\Rightarrow f(a) = g(x)$
 \therefore No unifier exists

iv) $\{\text{k}(\text{Sara}, n) \text{ and } \text{k}(\text{Sara}, y, \text{mother}(y))\}$
 $y \rightarrow \text{John}$ $\{y \rightarrow \text{John}\}$
 $n \rightarrow \text{mother}(y)$ $n \rightarrow \text{mother}(\text{John})$
 $\rightarrow \text{mother}(\text{John})$

```

import json

# --- Helper Functions for Term Manipulation ---

def is_variable(term):
    """Checks if a term is a variable (a single capital letter string)."""
    return isinstance(term, str) and len(term) == 1 and 'A' <= term[0] <= 'Z'

def occurs_check(variable, term, sigma):
    """
    Checks if 'variable' occurs anywhere in 'term' under the current substitution 'sigma'.
    This prevents infinite recursion (e.g., unifying X with f(X)).
    """
    term = apply_substitution(term, sigma) # Check the substituted term

    if term == variable:
        return True

    # If the term is a list (function/predicate), check its arguments recursively
    if isinstance(term, list):
        for arg in term[1:]:
            if occurs_check(variable, arg, sigma):
                return True

    return False

def apply_substitution(term, sigma):
    """
    Applies the current substitution 'sigma' to a 'term' recursively.
    """
    if is_variable(term):
        # If the variable is bound in sigma, apply the binding
        if term in sigma:
            # Recursively apply the rest of the substitutions to the binding's value
            # This is critical for chains like X/f(Y), Y/a -> X/f(a)
            return apply_substitution(sigma[term], sigma)

    return term

    if isinstance(term, list):
        # Apply substitution to the arguments of the function/predicate
        new_term = [term[0]] # Keep the function/predicate symbol
        for arg in term[1:]:
            new_term.append(apply_substitution(arg, sigma))
        return new_term

```

```

# Term is a constant or an unhandled type, return as is
return term

def term_to_string(term):
    """
    Converts the internal list representation of a term into standard logic notation string.
    e.g., ['f', 'Y'] -> "f(Y)"
    """
    if isinstance(term, str):
        return term

    if isinstance(term, list):
        # Term is a function or predicate
        symbol = term[0]
        args = [term_to_string(arg) for arg in term[1:]]
        return f'{symbol}({args})'

    return str(term)

# --- Main Unification Function ---

def unify(term1, term2):
    """
    Implements the Unification Algorithm to find the MGU for term1 and term2.
    Returns the MGU as a dictionary or None if unification fails.
    """
    # Initialize the substitution set (MGU)
    sigma = {}

    # Initialize the list of pairs to resolve (the difference set)
    diff_set = [[term1, term2]]

    print(f'--- Unification Process Started ---')
    print(f'Initial Terms:')
    print(f'L1: {term_to_string(term1)}')
    print(f'L2: {term_to_string(term2)}')
    print("-" * 35)

    while diff_set:
        # Pop the current pair of terms to unify
        t1, t2 = diff_set.pop(0)

        # 1. Apply the current MGU to the terms before comparison
        t1_prime = apply_substitution(t1, sigma)
        t2_prime = apply_substitution(t2, sigma)

```

```

print(f"Attempting to unify: {term_to_string(t1_prime)} vs {term_to_string(t2_prime)}")

# 2. Check if terms are identical
if t1_prime == t2_prime:
    print(f" -> Identical. Current MGU: {term_to_string(sigma)}")
    continue

# 3. Handle Variable-Term unification
if is_variable(t1_prime):
    var, term = t1_prime, t2_prime
elif is_variable(t2_prime):
    var, term = t2_prime, t1_prime
else:
    var, term = None, None

if var:
    # Check if term is a variable, and if so, don't bind V/V
    if is_variable(term):
        print(f" -> Both are variables. Skipping {var} / {term}")
        # Ensure they are added back if not identical (which is caught by step 2).
        # If V1 != V2, we add V1/V2 or V2/V1 to sigma. Since step 2 handles V/V, this means V1
        != V2 here.
        if var != term:
            sigma[var] = term
            print(f" -> Variable binding added: {var} / {term_to_string(term)}. New MGU:
{term_to_string(sigma)}")
            # Occurs Check: Fail if the variable occurs in the term it's being bound to
            elif occurs_check(var, term, sigma):
                print(f" -> OCCURS CHECK FAILURE: Variable {var} occurs in
{term_to_string(term)}")
                return None

            # Create a new substitution {var / term}
        else:
            sigma[var] = term
            print(f" -> Variable binding added: {var} / {term_to_string(term)}. New MGU:
{term_to_string(sigma)}")

# 4. Handle Complex Term (Function/Predicate) unification
elif isinstance(t1_prime, list) and isinstance(t2_prime, list):
    # Check functor/predicate symbol and arity (number of arguments)
    if t1_prime[0] != t2_prime[0] or len(t1_prime) != len(t2_prime):
        print(f" -> FUNCTOR/ARITY MISMATCH: {t1_prime[0]} != {t2_prime[0]} or arity
mismatch.")
        return None

```

```

# Add corresponding arguments to the difference set
# Start from index 1 (after the symbol)
for arg1, arg2 in zip(t1_prime[1:], t2_prime[1:]):
    diff_set.append([arg1, arg2])
print(f" -> Complex terms matched. Adding arguments to difference set.")

# 5. Handle Constant-Constant or other mismatches (Fail)
else:
    print(f" -> TYPE/CONSTANT MISMATCH: {term_to_string(t1_prime)} and
{term_to_string(t2_prime)} cannot be unified.")
    return None

print("-" * 35)
print("--- Unification Successful ---")

# Final cleanup to ensure all bindings are fully resolved
final_mgu = {k: apply_substitution(v, sigma) for k, v in sigma.items()}
return final_mgu

# --- Define the Input Terms ---

# L1 = Q(a, g(X, a), f(Y))
literal1 = ['Q', 'a', ['g', 'X', 'a'], ['f', 'Y']]

# L2 = Q(a, g(f(b), a), X)
literal2 = ['Q', 'a', ['g', ['f', 'b'], 'a'], 'X']

# --- Run the Unification ---

mgu_result = unify(literal1, literal2)

if mgu_result is not None:
    print("\n[ Final MGU Result ]")

    # Format the final MGU for display using the new helper function
    clean_mgu = {k: term_to_string(v) for k, v in mgu_result.items()}
    final_output = ', '.join([f'{k} / {v}' for k, v in clean_mgu.items()])
    print(f'Final MGU: {{ {final_output} }}')

# --- Verification ---
print("\n[ Verification ]")
unified_l1 = apply_substitution(literal1, mgu_result)
unified_l2 = apply_substitution(literal2, mgu_result)

print(f'L1 after MGU: {term_to_string(unified_l1)}')
print(f'L2 after MGU: {term_to_string(unified_l2)}')

```

```
if unified_l1 == unified_l2:  
    print("-> SUCCESS: L1 and L2 are identical after applying the MGU.")  
else:  
    print("-> ERROR: Unification failed verification.")  
else:  
    print("\nUnification FAILED.")
```

Program 8

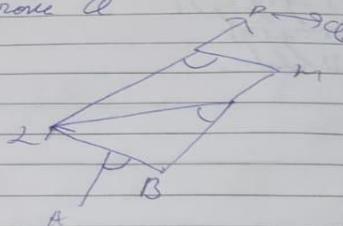
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning:

Forward Reasoning

First Order Logic

Create a KB consisting of FO statements and the given query using forward reasoning

$$\begin{array}{ll} 1) P \Rightarrow Q & S \models \text{fact} \\ 2) M = P & \text{Prove } Q \\ 3) L = M & \\ 4) P = L & \\ 5) A = B & \end{array}$$



- i) The law says
ii) The law says that it is a crime for an American to sell weapons to hostile nations. The country Nano, ally enemy of America, has some missiles, and all of its missiles were sold to it by colonel West, who is American. Prove that "West is criminal".

$$1) \forall n, y, z \text{ American}(n) \wedge \text{Weapon}(y) \wedge \text{Sells}(n, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Crimes}(n)$$

$$2) \forall n \text{ Missle}(n) \wedge \text{Owens}(Nano, n) \Rightarrow \text{Sells}(West, n,$$

- 3) $\forall x \text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$
 4) $\forall x \text{Mussel}(x) \Rightarrow \text{Weapon}(x)$
 5) $\text{American}(\text{Alice})$
 6) $\text{Enemy}(\text{None}, \text{America})$
 7) $\text{Owning}(\text{None}, \text{M1})$
 8) $\text{Mussel}(\text{M1})$

Algos

1) Initialize

- Add all known facts from KB to Agenda
- Set Inferred not empty

2) While Agenda not empty

- Remove the first fact f from Agenda
- If f matches (unifies with) the query, return

True

• If $f \notin \text{Inferred}$,

• Add f to Inferred

• For every fact K in Lits where f matches

some in the body of K :

• Use unification to find scales θ so $P, \theta = f$

• Apply θ to all other premises in R

• If all premises P_i , $- P_m$ of K are stable
return order θ .

3) If last finished without finding query return
False.

```

from typing import List, Tuple, Dict, Set, Union

Predicate = Tuple[str, Tuple[str, ...]]

class Rule:
    def __init__(self, head: Predicate, body: List[Predicate]):
        self.head = head
        self.body = body

    def __repr__(self):
        body_str = ', '.join(f'{p[0]} {p[1]}' for p in self.body)
        return f'{body_str} => {self.head[0]} {self.head[1]}'

# Knowledge base
class KnowledgeBase:
    def __init__(self):
        self.facts: Set[Predicate] = set()
        self.rules: List[Rule] = []

    def add_fact(self, fact: Predicate):
        self.facts.add(fact)

    def add_rule(self, rule: Rule):
        self.rules.append(rule)

    def forward_chain(self, query: Predicate) -> bool:
        inferred = set(self.facts)
        added = True

        while added:
            added = False
            for rule in self.rules:
                if all(self._match_fact(body_pred, inferred) for body_pred in rule.body):
                    if not self._match_fact(rule.head, inferred):
                        inferred.add(rule.head)
                        added = True
                        print(f'Inferred: {rule.head}')

                if self._match_fact(query, inferred):
                    return True
        return self._match_fact(query, inferred)

    def _match_fact(self, pred: Predicate, fact_set: Set[Predicate]) -> bool:
        return pred in fact_set

# --- Example usage ---
if __name__ == "__main__":

```

```

kb = KnowledgeBase()

kb.add_fact(("Parent", ("John", "Mary")))
kb.add_fact(("Parent", ("Mary", "Sue")))

facts_list = list(kb.facts)
for f1 in facts_list:
    for f2 in facts_list:
        if f1[0] == "Parent" and f2[0] == "Parent":
            if f1[1][1] == f2[1][0]:
                head = ("Grandparent", (f1[1][0], f2[1][1]))
                body = [f1, f2]
                kb.add_rule(Rule(head, body))

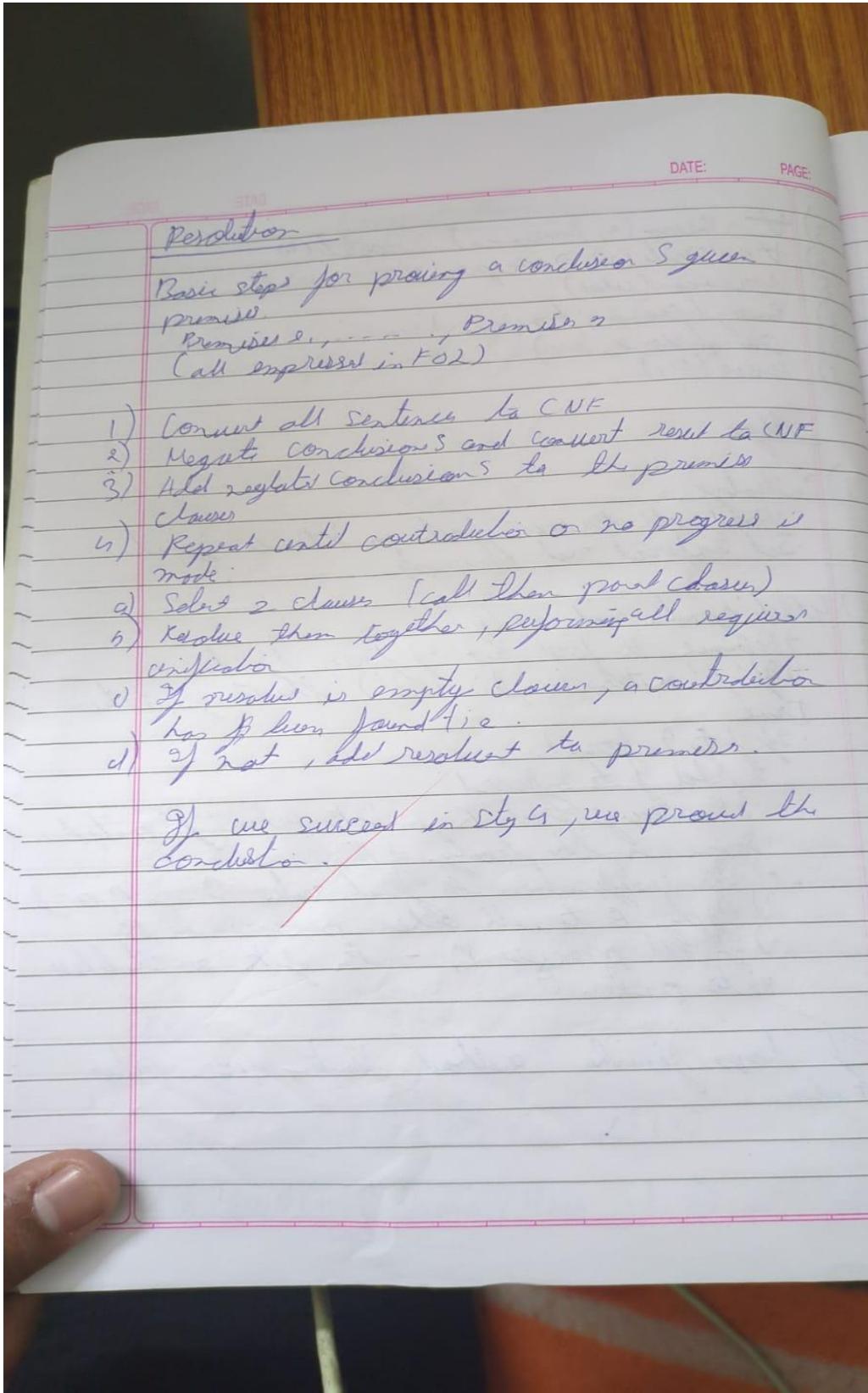
query = ("Grandparent", ("John", "Sue"))
result = kb.forward_chain(query)

print(f"Query {query} is", "True" if result else "False")

```

Program 9:

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution.



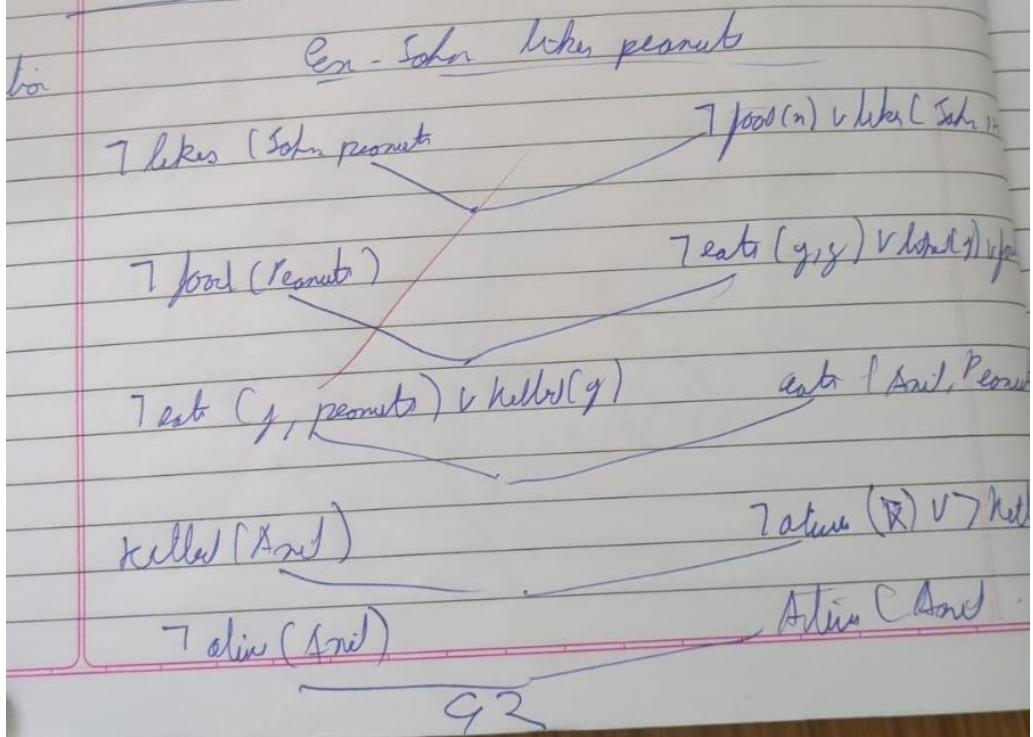
function minmax (state) return a action
 return argmax \in Action(s) min-value (Result [state])

function max - value (state) returns a utility value
 if Terminal - Test (state) return then return utility (state)
 action v

function Min - Value (state) return a utility value
 if terminal - Test (state) then return Utility (state)
 else

for each a in action (state) do
 $v \leftarrow \min (v, \text{Max - value} (\text{Result}' (s, a)))$

return v



```

from itertools import combinations

def pl_resolution(KB, query):
    # Negate the query and add to KB
    clauses = KB + [negate(query)]
    print("Initial Clauses:")
    for c in clauses:
        print(c)
    print("-" * 40)

    new = set()
    while True:
        # Generate all possible pairs of clauses
        pairs = list(combinations(clauses, 2))
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if [] in resolvents:
                print(f"Derived empty clause from {ci} and {cj}")
                return True
            new.update(tuple(sorted(r)) for r in resolvents)

        if new.issubset(set(tuple(sorted(c)) for c in clauses)):
            # No new clauses added — cannot derive contradiction
            return False
        for c in new:
            if list(c) not in clauses:
                clauses.append(list(c))

def resolve(ci, cj):
    """Resolve two clauses and return the resolvents."""
    resolvents = []
    for di in ci:
        for dj in cj:
            if di == negate(dj):
                new_clause = list(set(ci + cj))
                new_clause.remove(di)
                new_clause.remove(dj)
                resolvents.append(new_clause)
    return resolvents

def negate(literal):
    """Negate a literal."""
    if literal.startswith('~'):
        return literal[1:]
    else:

```

```

return '~' + literal

KB = [
    ['~R', 'W'],
    ['~W', 'G'],
    ['R']
]
query = 'G'

# --- Run Resolution ---
entailed = pl_resolution(KB, query)
print("\nResult:")
if entailed:
    print(f"↙ The knowledge base entails {query}.")
else:
    print(f"✗ The knowledge base does NOT entail {query}.")

```

Program 10:

Implement Alpha-Beta Pruning

DATE: PAGE:

→ Alpha - Beta search

Algo

function ALPHA-BETA-SEARCH(s_{lab}):
 value = MAX-VALUE(s_{lab} , $-s$, s)
 return the action from ACTIONS(s_{lab}) that
 produced value.

function MAX-VALUE(s_{lab} , s , β):
 if TERMINAL-TEST(s_{lab}):
 return UTILITY(s_{lab})
 value = $-\infty$
 for each action in ACTIONS(s_{lab}):
 value = max(value, MIN-VALUE($s_{lab}, action, s, \beta$))
 if value $\geq \beta$:
 return value
 α = min(α , value)
 return value

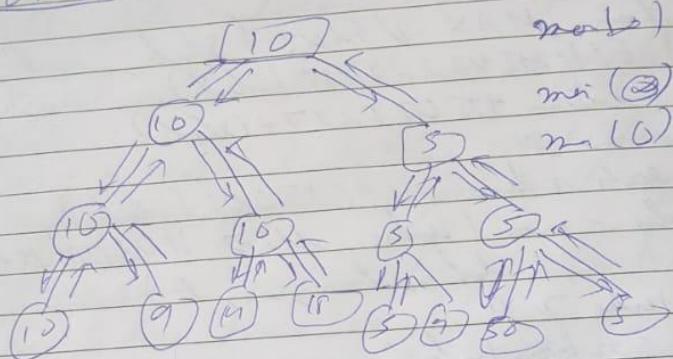
function MIN-VALUE(s_{lab} , s , α):
 if TERMINAL-TEST(s_{lab}):
 return UTILITY(s_{lab})
 value = $+\infty$
 for each action in A(TONS(s_{lab})): β
 value = min(value, MAX-VALUE($s_{lab}, action, -s, \alpha$))

if value of ζ :
other value-

$$\beta = \text{min}(\beta, \text{value})$$

retire value

Min - max



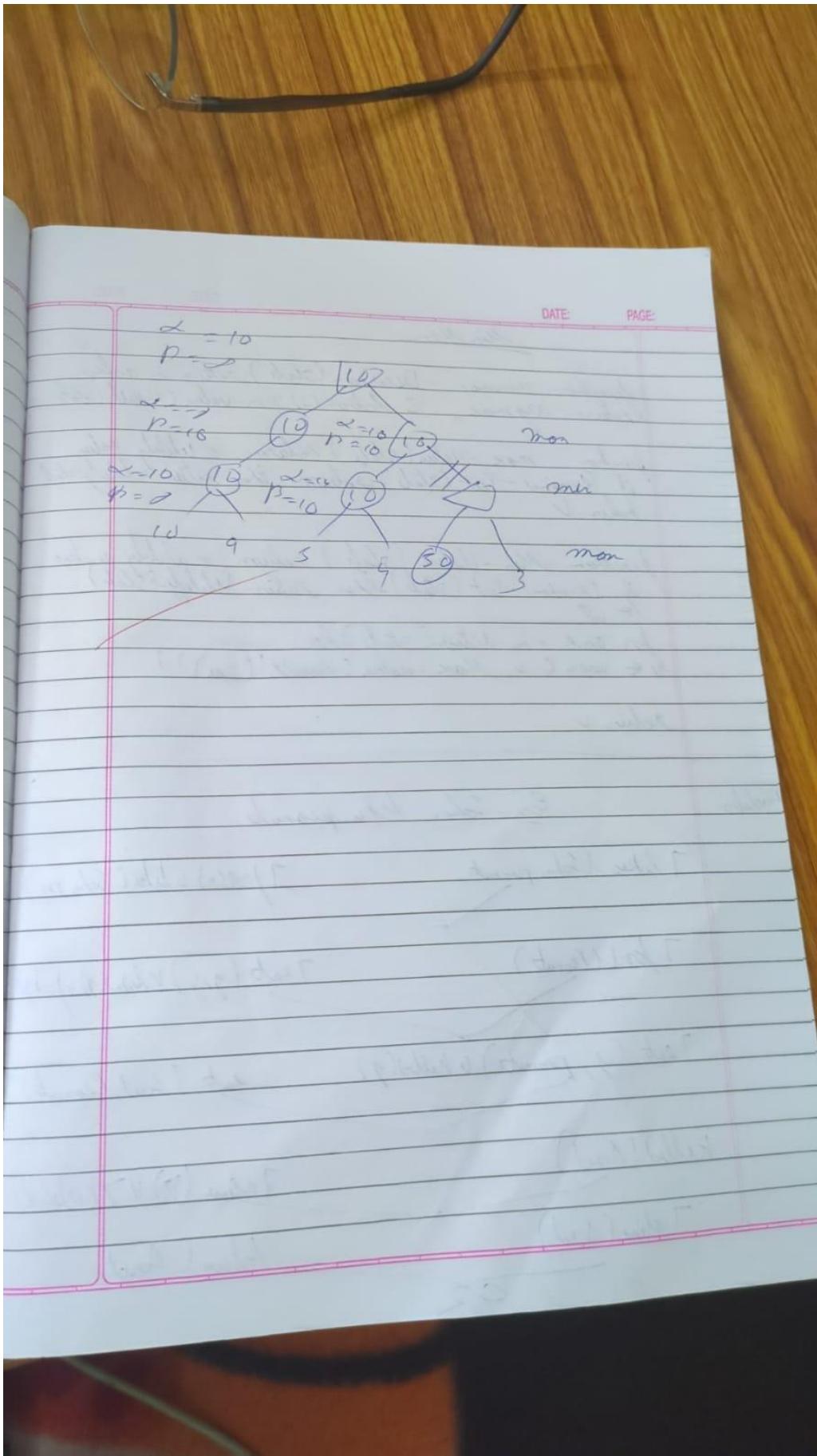
2001)

mei (B)

m (0)

$L = -\infty$
 $P = +\infty$
 $t = -\infty$
 $R = P_0$
 $L = -17$
 $D = -10$

max (λ)
 Bob (P)
 min (λ)



```

# --- Alpha-Beta Pruning Implementation ---

import math

# Minimax with Alpha-Beta Pruning
def alphabeta(depth, node_index, maximizing_player, values, alpha, beta, max_depth):
    """
    depth: current depth in the game tree
    node_index: index of the current node in the list of values
    maximizing_player: boolean, True if it's the maximizing player's turn
    values: list of terminal node values
    alpha, beta: alpha-beta values for pruning
    max_depth: maximum depth of the tree
    """
    # Base case: if we reach the leaf node
    if depth == max_depth:
        return values[node_index]

    if maximizing_player:
        best = -math.inf
        # Explore left and right child
        for i in range(2):
            val = alphabeta(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            # Prune
            if beta <= alpha:
                break
        return best
    else:
        best = math.inf
        # Explore left and right child
        for i in range(2):
            val = alphabeta(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)
            # Prune
            if beta <= alpha:
                break
        return best

# --- Example Game Tree ---

# Terminal node values (leaf nodes)
values = [3, 5, 6, 9]

```

```
# Tree depth = log2(len(values)) = 2
max_depth = 2

# Run alpha-beta pruning
best_value = alphabeta(0, 0, True, values, -math.inf, math.inf, max_depth)
print(f"The optimal value is: {best_value}")
```