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 **Anshuman 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

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

Infosys®
Navigate your next

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!

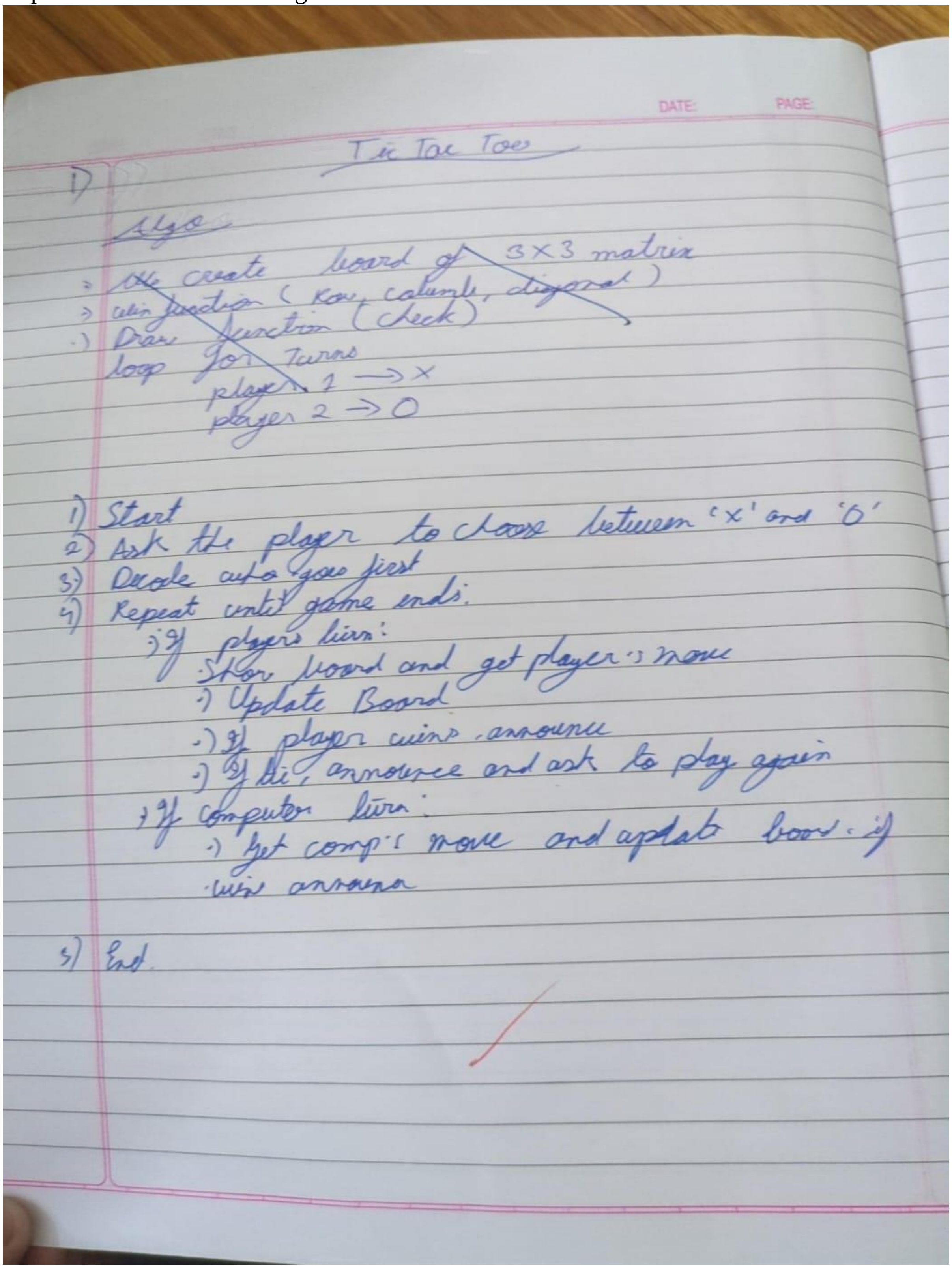
Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com

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



Tic Tac Toe

1)

Algo

- We create board of 3×3 matrix
- win function (Row, column, diagonal)
- Draw function (check)

loop for Turns

player 1 → X

player 2 → O

1) Start
2) Ask the player to choose between 'x' and 'O'
3) Decide who goes first
4) Repeat until game ends:
   - If player's turn:
     - Show board and get player's move
     - Update Board
     - If player wins, announce
     - If tie, announce and ask to play again
   - If computer turn:
     - Get comp's move and update board. if win announce
5) End.

Pseudocode

Create board of 3x3 matrix

def print_board (board): display the board in 3x3 grid

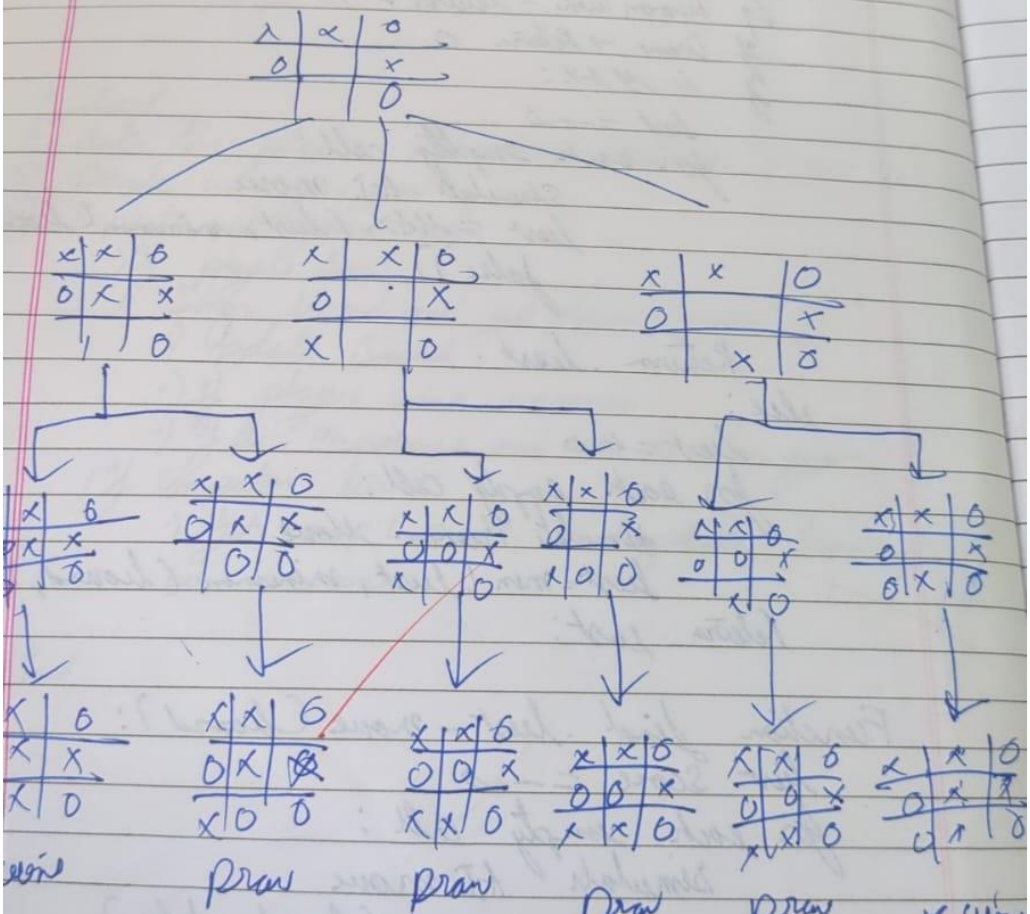
```
function minimax (board, is_max):
    if AI wins - return +1
    if human wins - return -1
    if Draw -> Return 0
    if is MAX:
        best = -∞
        for each empty cell:
            simulate AI move
            best = MAX (best, minimax (board, false))
        Return best.
    else:
        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)
        undo move
```

Algorithm:

If score > best.score
best.score = score
best.move = cell.

Output

Initial state : X



DATE: PAGE:

Vacuum Cleaner

Algo:

1) Divide environment into 4 quadrants (a1–a4)
2) Assign each quadrant randomly as clean/dirty.
3) Start from Q1
4) at each step:
   if dirt → clean
   ?. else → skip
   ?) Move clockwise to next quadrant
5) Repeat for required cycles.
6) print final status.

Out Put

Action: one of {L, R, D, U, S}
while there exist at least one Dirty cell
in grid.

x, y ← position
check grid [x] [y] is Dirty then
perform cleaning
if not
if there is an dirty cell adjacent.
then choose a direction (L, R, U, D).
toward that cell.
Move to the new position.
else
Move row by row to cover all cells.
stop output.

DATE: PAGE:

$\left|\begin{smallmatrix} V & D \\ D & \end{smallmatrix}\right| \rightarrow \left|D \; V\right| \rightarrow \left|D \; V\right| \rightarrow \left|V\right|$

Room is clean!

<u>Pseudocode</u>

```
class vacuumcleaner:
  INIT:
    quadrant = [Q1, Q2, Q3, Q4]
    status = random(clean / dirty) for each
        quadrant)
    current_index = 0

  Function clean (q)
    mark q as clean

  Function move_next ():
        move clockwise to next quadrant.

  Function start (cycles)
    Repeat cycles x4:
        current = quadrants [current_index]
        if status [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

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")
```

vaccum cleaner:

```
rooms = int(input("Enter Number of rooms: "))
Rooms = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
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 1 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 PUZZLE (~~Misplaced~~)

~~Algo~~ Pseudocode

~~1) Start with initial state.~~

FUNCTION AStar (start_state, goal_state):

DEFINE priority_queue PQ
INSERT (f = heuristic(start_state), g=0, state = start_state, path=[]) INTO PQ.

DEFINE visited_set = {}

WHILE PQ is not ~~empty~~ empty:
(f, g, state, path) = remove smallest f from PQ

IF state is in visited_set:
Continue
ADD state TO visited_set:

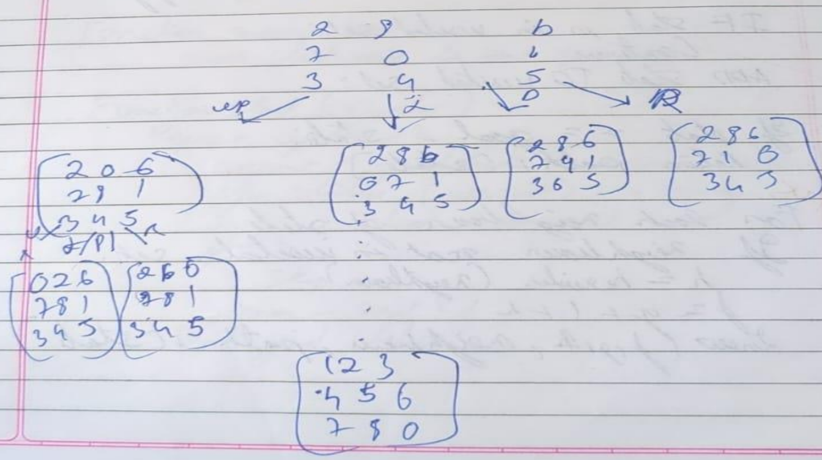
If state == goal_state:
Return path + [state]

For each neighbour of state
If neighbour not in visited_set:
h = heuristic (neighbour)
f = g + 1 + h
Insert (f, g+1, neighbour, path + [state])

Function heuristic (state):
    count = 0
    for each tile at position (i, j):
        if tile != 0 & tile != goal_state [i] [j]
            count = count + 1
    return count

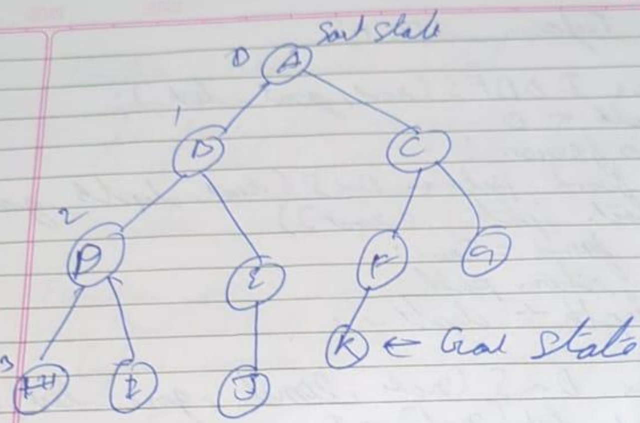
Function get_neighbours (state):
    Find blank position (x, y)
    Define moves = [(up), (down), (left), (right)]

    For each valid move:
        swap blank with adjacent tile
        Add new state 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_path):
    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, B, 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, J, C, F, K, G
Goal found.

The goal state is found at depth 3

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 the blank):
   if tile is not in the 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		8
4	2	1
7	3	6

Initial 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
```

```
        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:

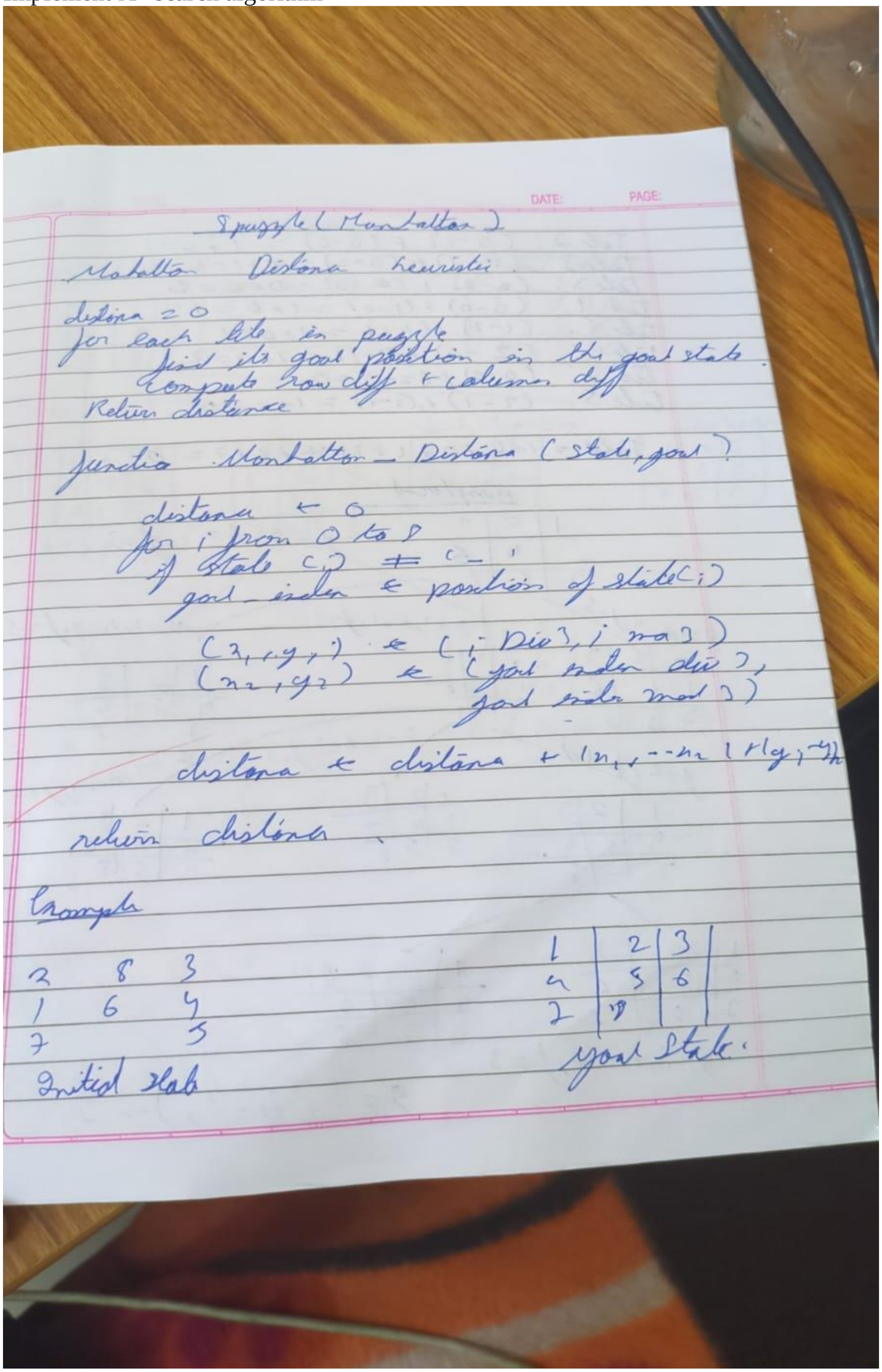
```
goal_state = '123456780'

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



8 puzzle (Manhattan)

Manhattan Distance heuristic.

distance = 0
for each tile in puzzle
  find its goal position in the goal state.
  compute row diff + column diff
Return distance

function Manhattan_Distance (state, goal)

  distance ← 0
  for i from 0 to 8
   if state (i) ≠ '_'
   goal_index ← position of state(i)

   (x_1, y_1) ← (i Div 3, i mod 3)
   (x_2, y_2) ← (goal index div 3,
          goal index mod 3)

   distance ← distance + $|x_1 - x_2| + |y_1 - y_2|$

return distance.

Example

2	8	3
1	6	4
7		5

Initial state

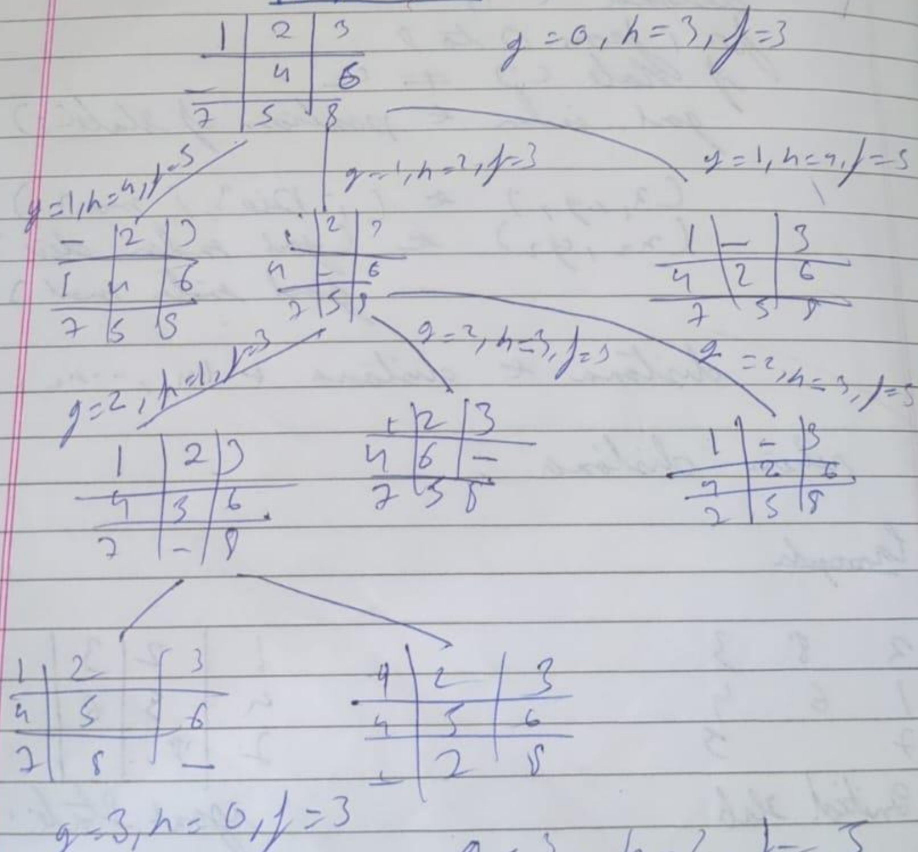
1	2	3
4	5	6
7	8	

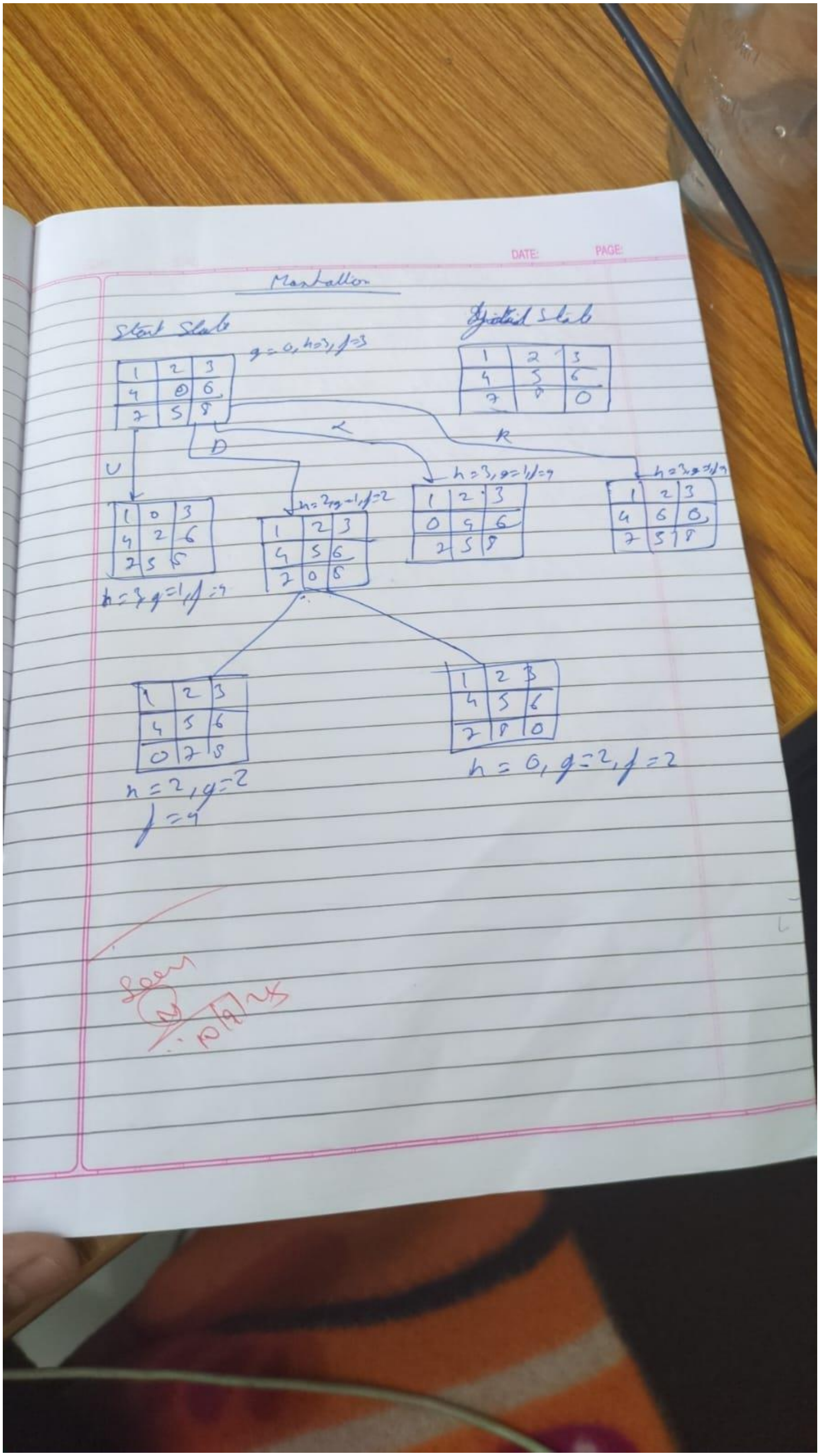
goal state.

Tile 2 : (0-1) + (0-0) = 1+0=1
Tile 8 : (1-1) + (0+2) = 0+2=2
Tile 3 : (0-0) + (0-0) = 0+0=0
Tile 1 : (0-0) + (1-0) = 1+0 = 1
Tile 6 : (1-2) + (1-1) = 1+0=1
Tile 4 : (2-0) + (1-1) = 2+0 = 2
Tile 7 : (0-0) + (2-2) = 0
Tile 5 : (2-1) + (2-1) = 1+1 = 6

Total = 1+2 +0 + 1+1+2 +2+2 = 9

misplaced




DATE:
PAGE:
Manhattan
Start State
g = 0, h=3, f=3
U
D
L
R
h = 3, g = 1, f = 4
h = 2, g = 1, f = 2
h = 3, g = 1, f = 4
h = 2, g = 2
f = 4
h = 0, g = 2, f = 2

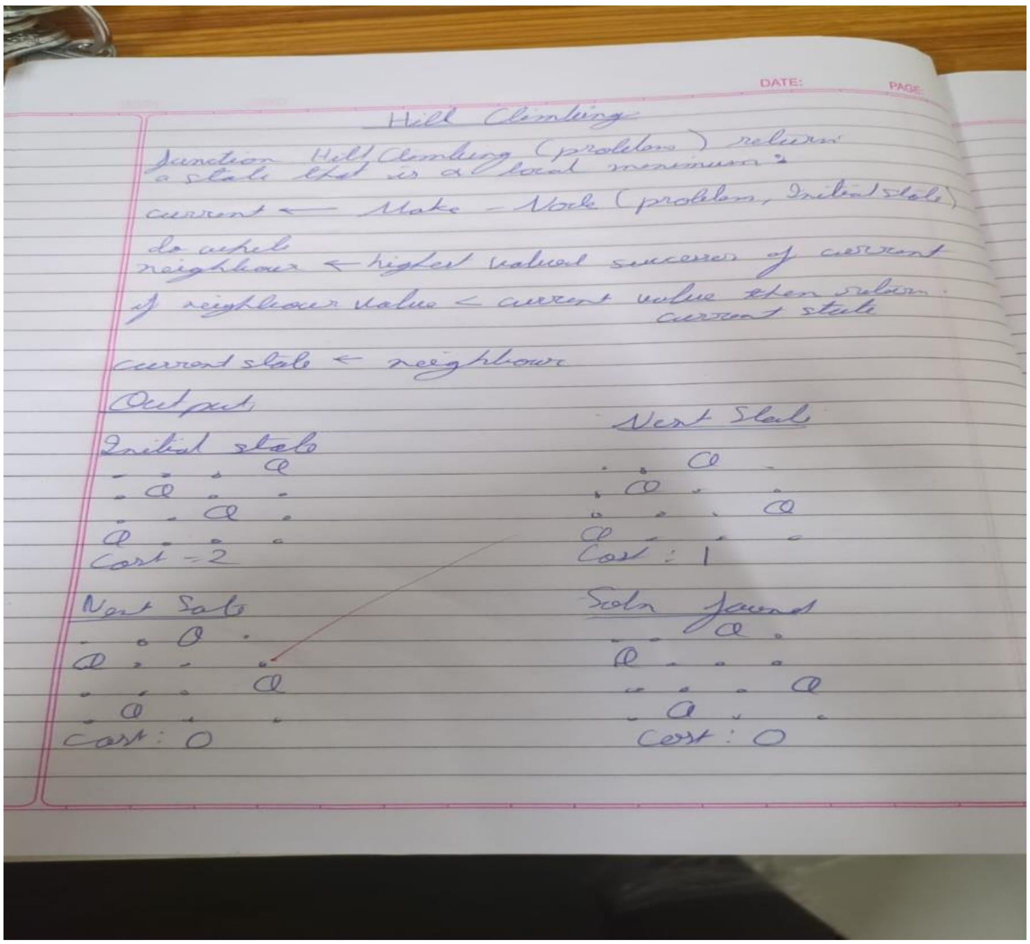
```
import heapq

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



Hill Climbing

function Hill Climbing (problem) returns a state that is a local maximum

current ← Make-Node (problem, Initial state)

do while

neighbour ← highest valued successor of current

if neighbour value < current value then return current state

current state ← neighbour

Output

Initial state

. . . Q
. Q . .
. . Q .
Q . . .

Cost = 2

Next State

. . Q .
. Q . .
. . . Q
Q . . .

Cost : 1

Next Sate

. . Q .
Q . . .
. . . Q
. Q . .

Cost : 0

Soln found

. . Q .
Q . . .
. . . Q
. Q . .

Cost : 0

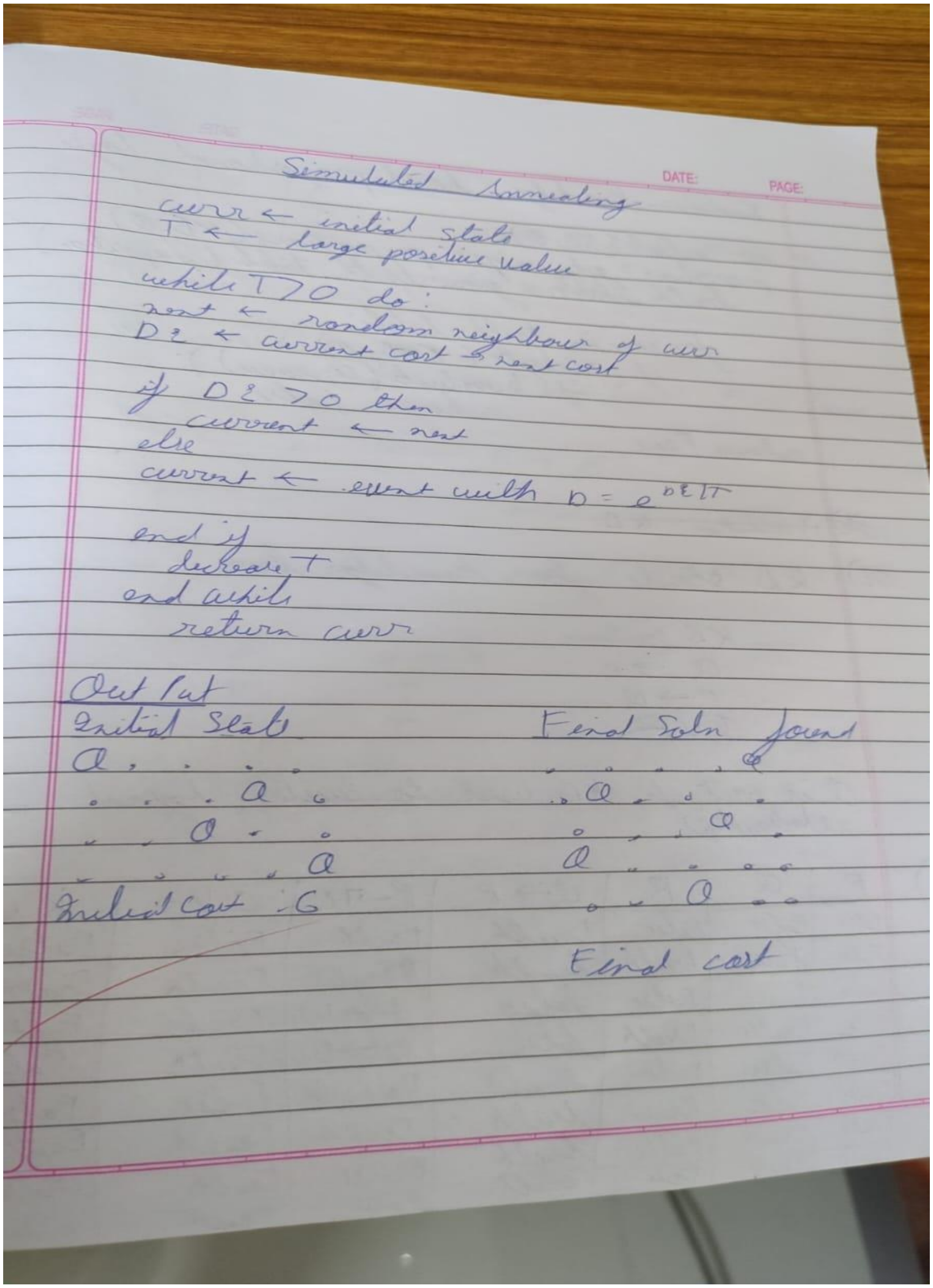
```
import random
import time

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

```python
    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


Simulated Annealing

curr ← initial state
T ← large positive value

while T > 0 do:
next ← random neighbour of curr
DE ← current cost - next cost

if DE > 0 then
current ← next
else
current ← event with $p = e^{DE/T}$

end if
decrease T
end while
return curr

Output
Initial State

Q . . . .
. . . Q .
. . Q . .
. . . . Q

Initial cost : 6

Final Soln found

. . . . Q
. Q . . .
. . . Q .
Q . . . .
. . Q . .

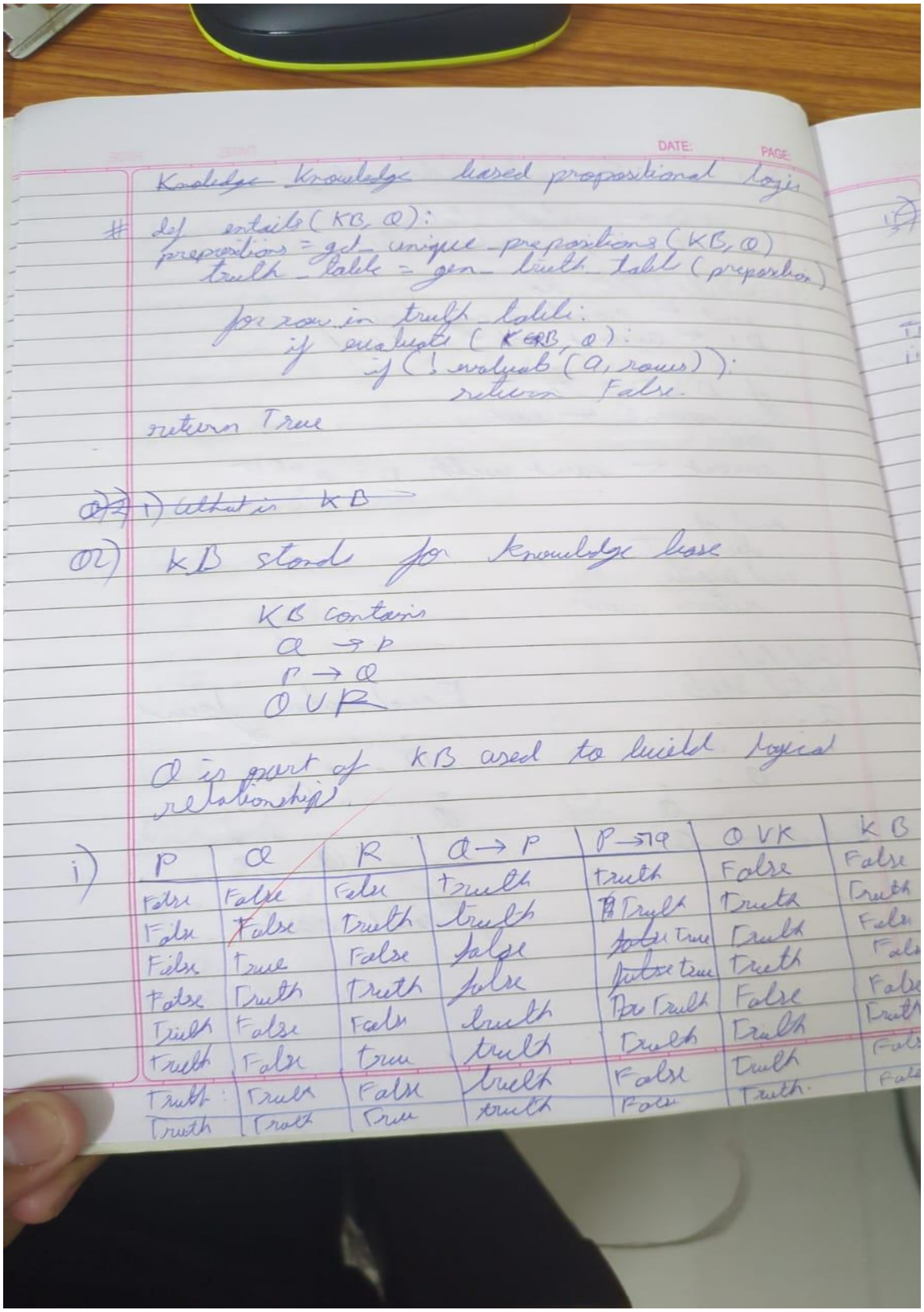
Final cost

```python
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_neighbor(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_neighbor(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:



~~Knoledge~~ Knowledge based propositional logic

```
# def entails(KB, Q):
prepositions = get_unique_prepositions(KB, Q)
    truth_table = gen_truth_table(prepositions)

    for row in truth_table:
        if evaluate(KB, Q):
            if (!evaluate(Q, rows)):
                return False.
return True
```

~~Q1) 1) What is KB~~

Q2) KB stand for knowledge base

KB contains

$Q \rightarrow P$

$P \rightarrow Q$

$Q \vee R$

Q is part of KB used to build logical relationship.

i)

P	Q	R	Q → P	P → ¬q	Q V R	KB
False	False	False	truth	truth	False	False
False	False	Truth	truth	Truth	Truth	Truth
False	True	False	false	~~false~~ true	Truth	False
False	Truth	Truth	false	~~false~~ true	Truth	False
Truth	False	False	truth	Truth	False	Truth
Truth	False	true	truth	Truth	Truth	False
Truth	Truth	False	truth	False	Truth	False
Truth	Truth	True	truth	False	Truth.	[illegible]

DATE PAGE

i)

P	Q	R
F	F	T
T	F	T

~~KB entails R~~

These are the two models where R is true

ii) KB entails R

Since whenever KB is true, R is true

iii)

P	R	R → P
F	T	F
T	T	T

So, KB, does not entail R → P

iv)

Q	R	Q → R
F	T	T
F	T	T

So, KB ~~does not~~ entails Q → R

proceed

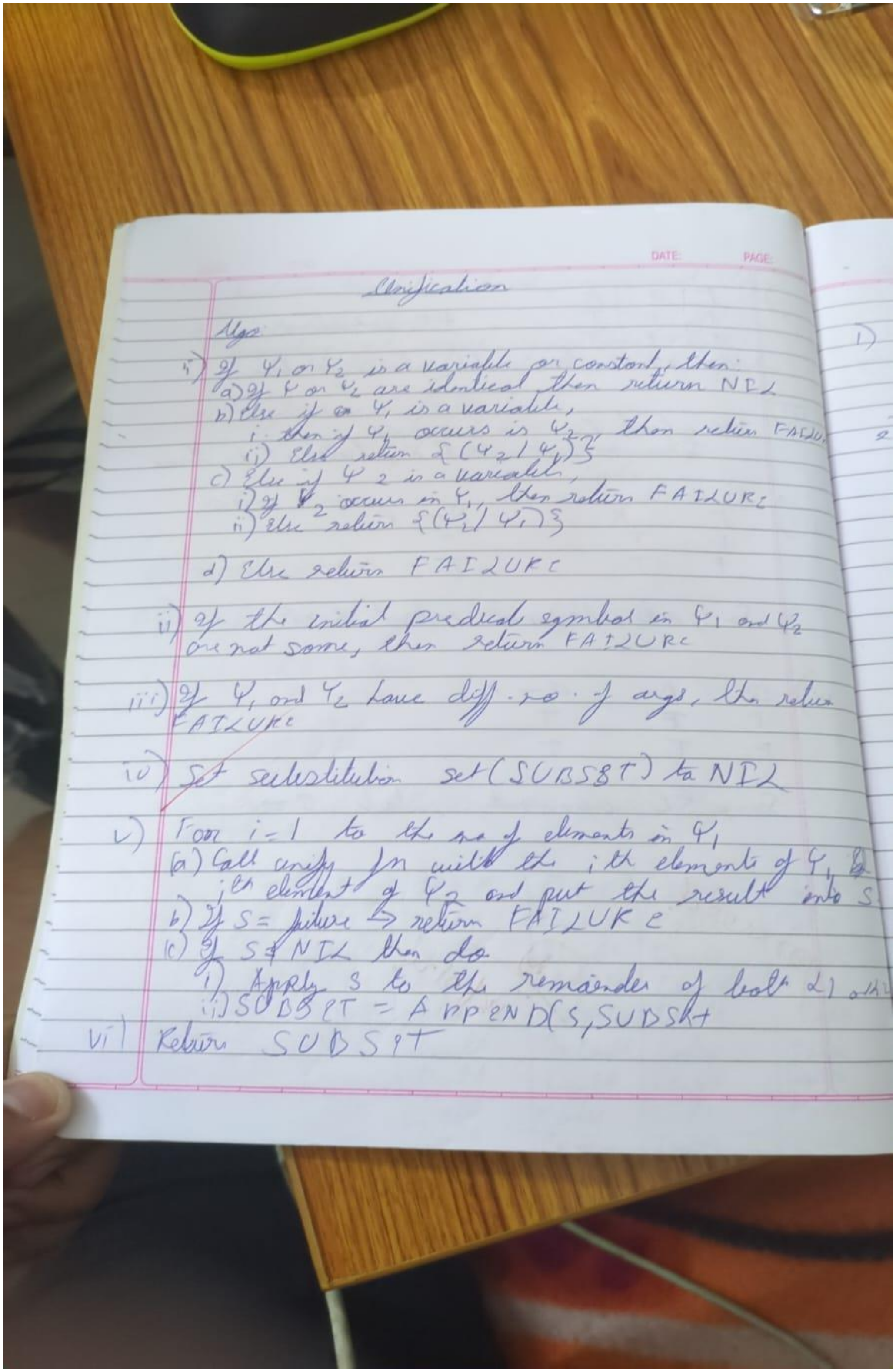
18/10/25

```python
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:



Unification

Algo:

i) If Ψ_1 or Ψ_2 is a variable or constant, then:
a) If Ψ_1 or Ψ_2 are identical then return NIL
b) Else if Ψ_1 is a variable,
i. then if Ψ_1 occurs in Ψ_2, then return FAILURE
ii) Else return $\{(\Psi_2/\Psi_1)\}$
c) Else if Ψ_2 is a variable,
i) If Ψ_2 occurs in Ψ_1, then return FAILURE
ii) Else return $\{(\Psi_1/\Psi_2)\}$

d) Else return FAILURE

ii) If the initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE

iii) If Ψ_1 and Ψ_2 have diff. no. of args, then return FAILURE

iv) Set substitution set(SUBSET) to NIL

v) For i = 1 to the no of elements in Ψ_1
(a) Call unify fn with the ith element of Ψ_1 & ith element of Ψ_2 and put the result into S
b) If S = failure => return FAILURE
(c) If S ≠ NIL then do
i) Apply S to the remainder of both L1 and L2
ii) SUBSET = APPEND(S, SUBSET)

vi) Return SUBSET

eg:-

1) Unify (knows (John, x), knows(John, Jane))
$\theta = x / Jane$
Unify (knows (John, Jane), knows (John, Jane))

2) Unify (knows (John, x), knows(y, B'r11))
$\theta = y / John$

Unify (knows (John, Bx), knows (John, Bτ11
$\theta = x / B\tau 11$

Unify (knows (John, B11), Knows (John, B

i) Check for {p(b, x, f(g(z))) and p(z, f(i), f
~~{z → f(b), y → b~~
{z → b, x → f(i) i → g(z)}

ii) {p(a, g(x, x), f(y)) and p(a, g(f(b), a), x)
{x → f(b), y → b}

iii) {p(f(a), g(x)), f(x, y)}
X = f(a) and X = g(x)
→ f(a) = g(x)
∴ No unifier exists

iv) {knows(John, x) and knows (y, mother(y))}
y → John
x → mother (y)
→ mother (John)

{y → John
x → mother (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}({', '.join(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

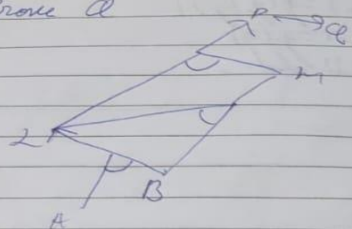
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning:

Forward Reasoning

First Order Logic

Create a KB consisting of FOL statement and the given query using forward reasoning

) $P \Rightarrow Q$ A } fact
$L \wedge M \Rightarrow P$ B
$B \wedge L \Rightarrow M$ Prove Q
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$



ii) ~~The Law Says~~
ii) The law ~~does~~ says that it a crime for an American to sell weapon to hostile nations. The country Nono, all 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) $\forall x, y, z$ American(x) ^ Weapon(y) ^ Sells(x, y, z) ^ Hostile(z) ⇒ Criminal(x)

2) $\forall x$ Missile(x) ^ Owns(Nono, x) ⇒ Sells(West, x,

DATE: PAGE:

3) $\forall x$ Enemy (x, America) $\Rightarrow$ Hostile (x)
4) $\forall x$ Missile (x) $\Rightarrow$ Weapon (x)
5) American (West)
6) Enemy (Nono, America)
7) Owns (Nono, M1)
8) Missile (M1)

Algo

1) Initialize
· Add all known facts in KB to Agenda
· Set Inferred not empty

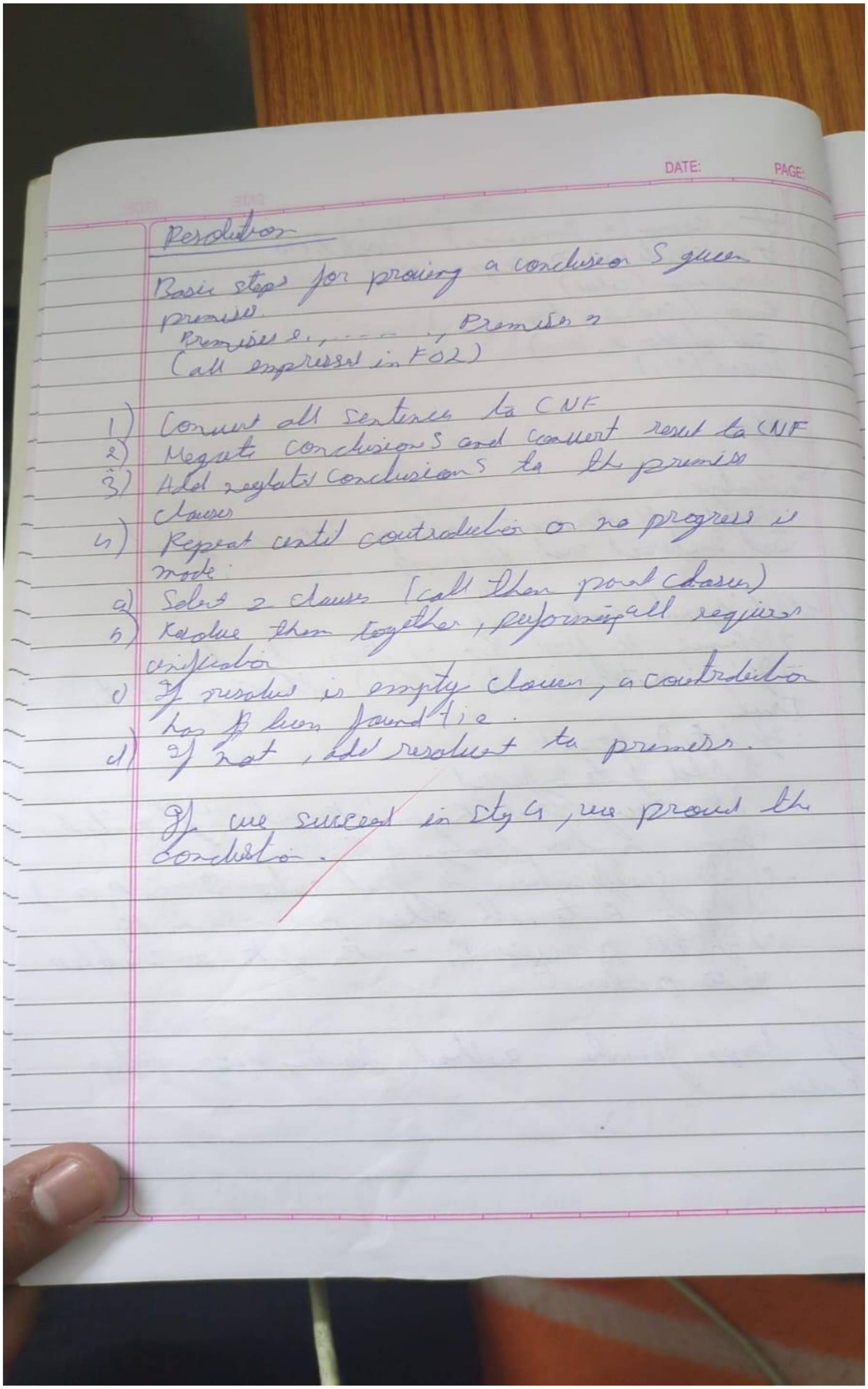
2) While Agenda not empty
·) Remove the first fact 'f' from Agenda
·) If 'f' matches (unifies with) the Query, return True
· If f ∉ Inferred.
·) Add f to Inferred
·) For every rule R in KB where 'f' matches some in the body of R:
·) Use unification to find subs θ so $P_i \theta = f$
·) Apply θ to all other premises in R
·) If all premises $P_1, \ldots P_n$ of R are satisfied extra order θ.

3) If loop finishes 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.



Resolution

Basic steps for proving a conclusion S given premises
Premise 1, ..., Premise n
(all expressed in FOL)

1) Convert all sentences to CNF
2) Negate conclusion S and convert result to CNF
3) Add negated conclusion S to the premise clauses
4) Repeat until contradiction or no progress is made:
a) Select 2 clauses (call them parent clauses)
b) Resolve them together, performing all required unification
c) If resolvent is empty clause, a contradiction has been found i.e.
d) If not, add resolvent to premises.

If we succeed in step 4, we proved the conclusion.

function minimax - Decision (state) return an action
return argmax$_a$ ∈ Action(s) Min-value (Result (state, a))

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

function Min - Value (state) return a utility value
if terminal - Test (state) then return Utility (State)
v ← ∞
for each a in action (state) do
v ← min (v, Max - value (Result (s,a)))

return v

Ex - John likes peanut

¬likes (John, peanuts) — ¬food (x) ∨ likes (John, x)

¬food (Peanuts) — ¬eats (y, z) ∨ killed (y) ∨ food (z)

¬eats (y, peanuts) ∨ killed (y) — eats (Anil, Peanuts)

killed (Anil) — ¬alive (x) ∨ ¬killed (x)

¬alive (Anil) — Alive (Anil)

93

```
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

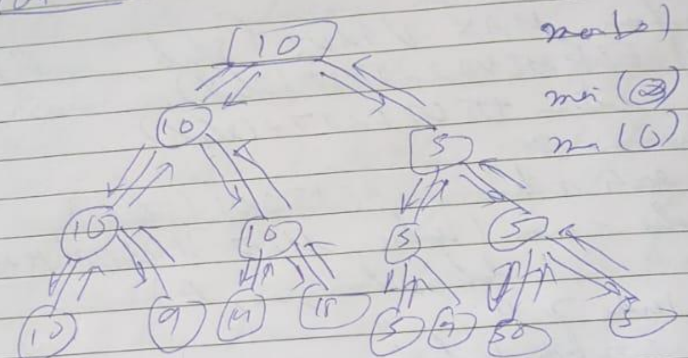
```
function ALPHA-BETA-SEARCH(state):
    value = MAX-VALUE(state, -∞, +∞)
    return the action from ACTIONS(state) that
    produced value.

function MAX-VALUE(state, α, β):
    if TERMINAL-TEST(state):
        return UTILITY(state)
    value = -∞
    for each action in ACTIONS(state):
        value = max(value, MIN-VALUE(RESULT)
                (state, action, α, β)
        if value ≥ β:
            return value

        α = max(α, value)
    return value

function MIN-VALUE(state, α, β)
    if TERMINAL-TEST(state):
        return UTILITY(state)
    value = +∞
    for each action in ACTIONS(state):
        value = min(value, MAX-VALUE(RESULT)(
        state, action), α, β)
```

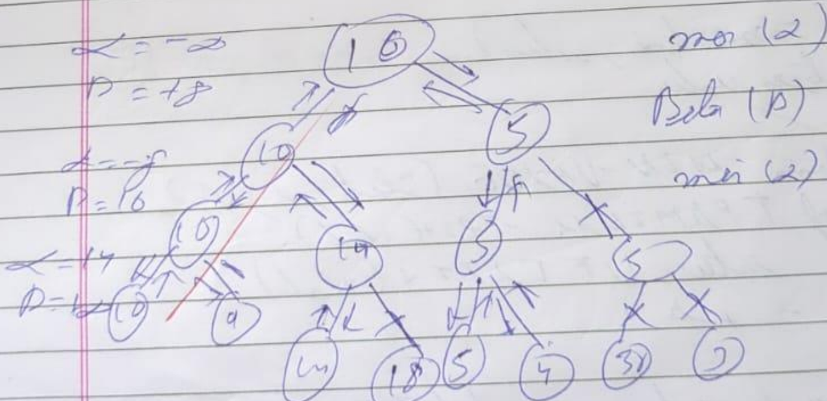
if value < α:
    return value.

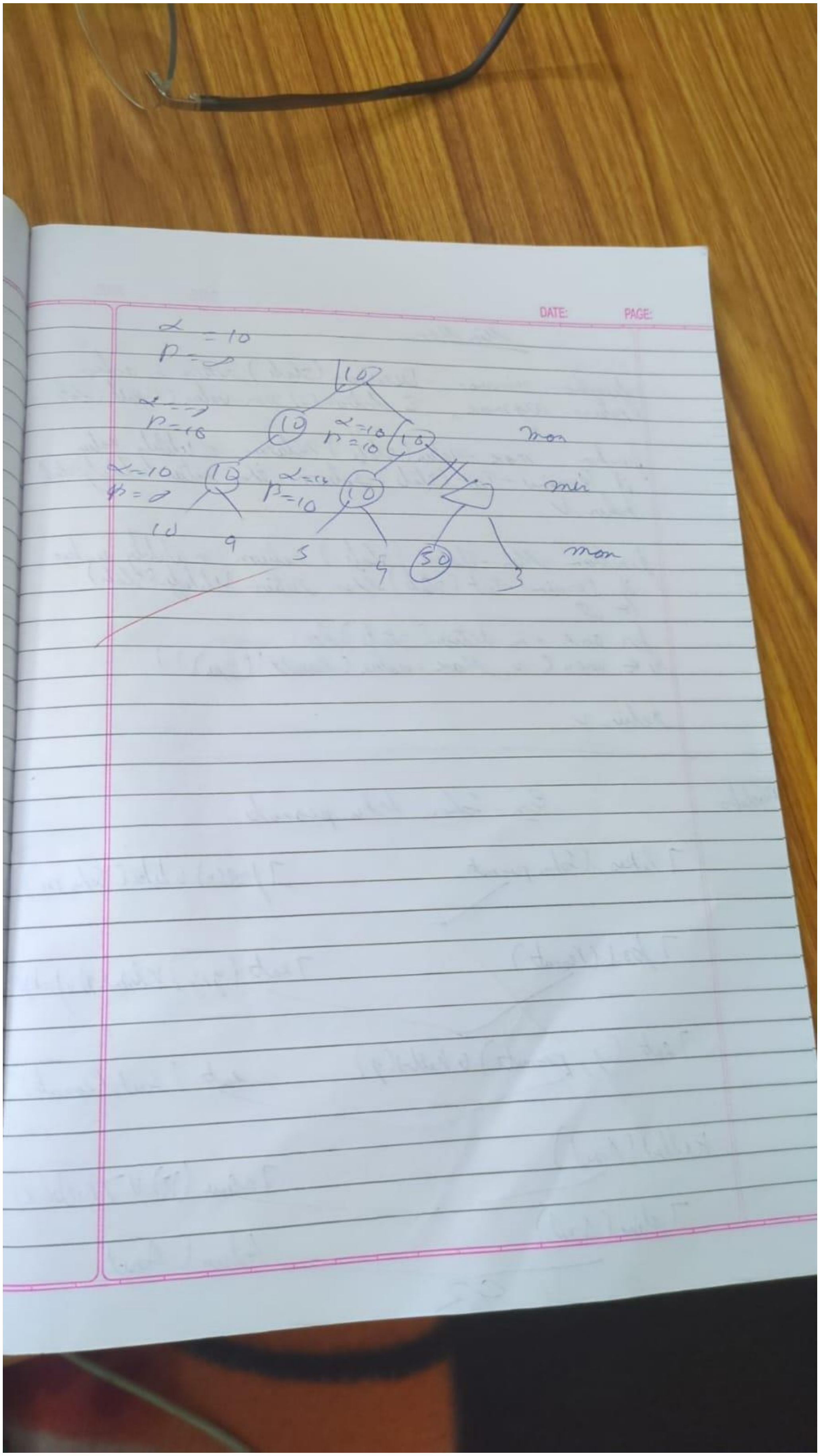
β = min(β, value)

return value

<u>Min-max</u>






DATE:
PAGE:
α = 10
β = ∞
α = -∞
β = 10
α = 10
β = ∞
α = 10
β = 10
α = 10
β = 10
10
max
min
max
10
9
5
4
50
3

```
# --- 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}")
```