VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shreyas Gowda C(1BM23CS319)

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

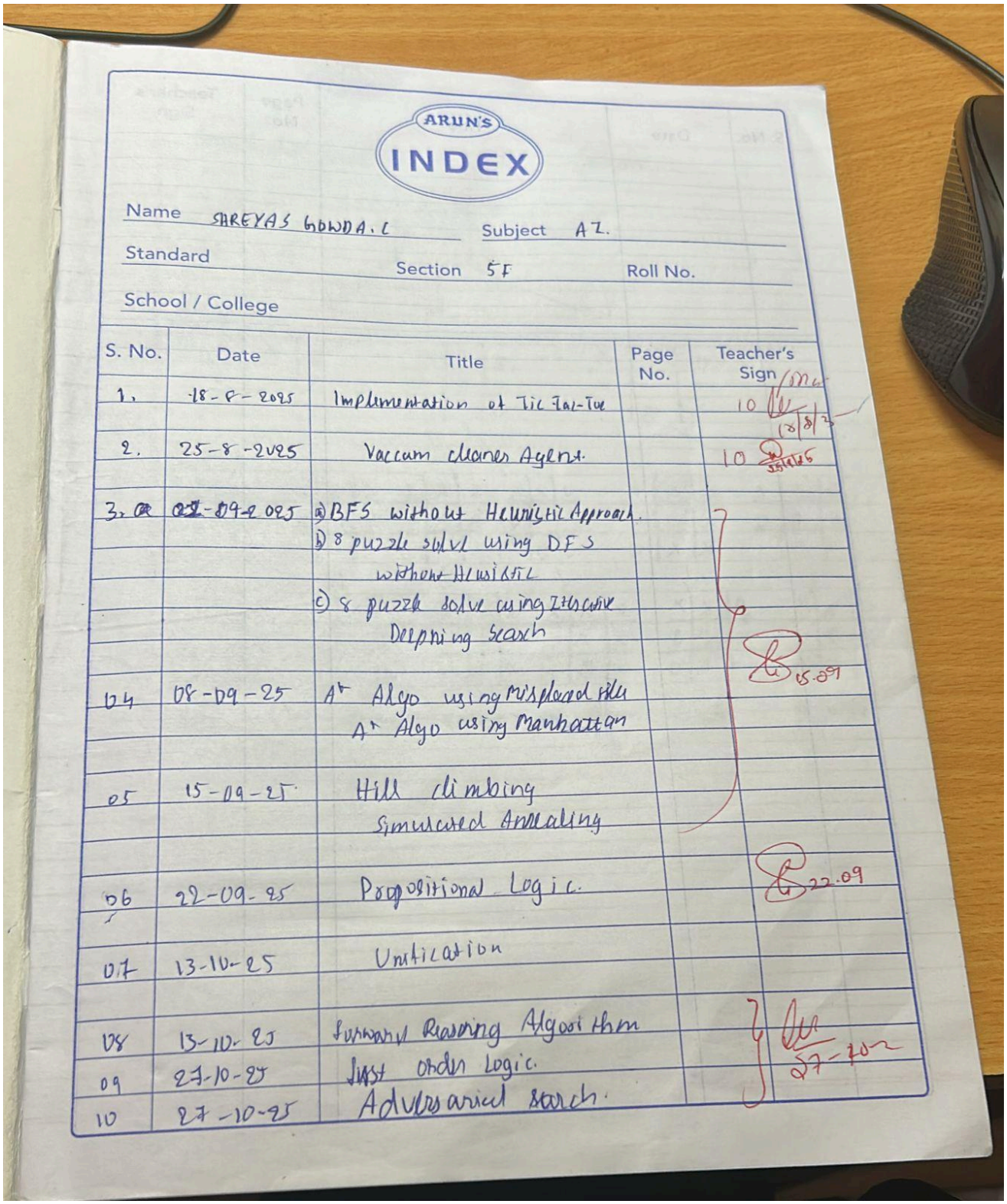
B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Shreyas Gowda C (1BM23CS319),** 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.

Lab faculty Incharge Name Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index



ARUN'S

INDEX

Name SHREYAS GOWDA. C Subject AI.

Standard Section 5F Roll No.

School / College

S. No.	Date	Title	Page No.	Teacher's Sign
1.	18-8-2025	Implementation of Tic Tac-Toe		10
2.	25-8-2025	Vaccum cleaner Agent.		10
3.	01-09-2025	a) BFS without Heuristic Approach. b) 8 puzzle solve using DFS without Heuristic c) 8 puzzle solve using Iterative Deepning search		15.09
04	08-09-25	A* Algo using Misplaced tiles A* Algo using Manhattan		
05	15-09-25	Hill climbing Simulated Annealing		
06	22-09-25	Propositional Logic.		22.09
07	13-10-25	Unification		
08	13-10-25	Forward Reasoning Algorithm		27-10-25
09	27-10-25	First order Logic.		
10	27-10-25	Adversarial search.		

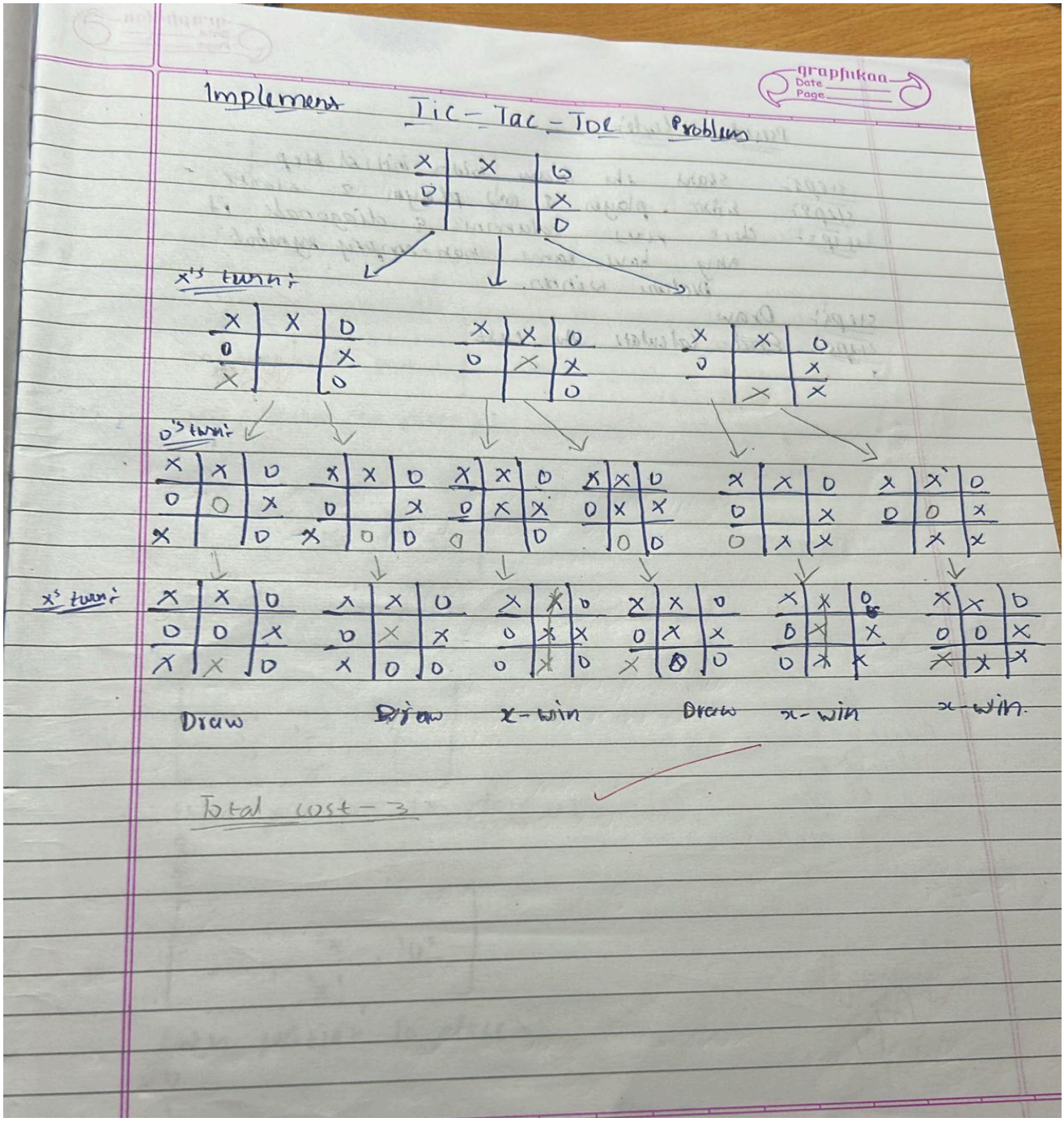
Github Link:
https://github.com/shreyasgowdac-319/1BM23CS319-AI-LAB

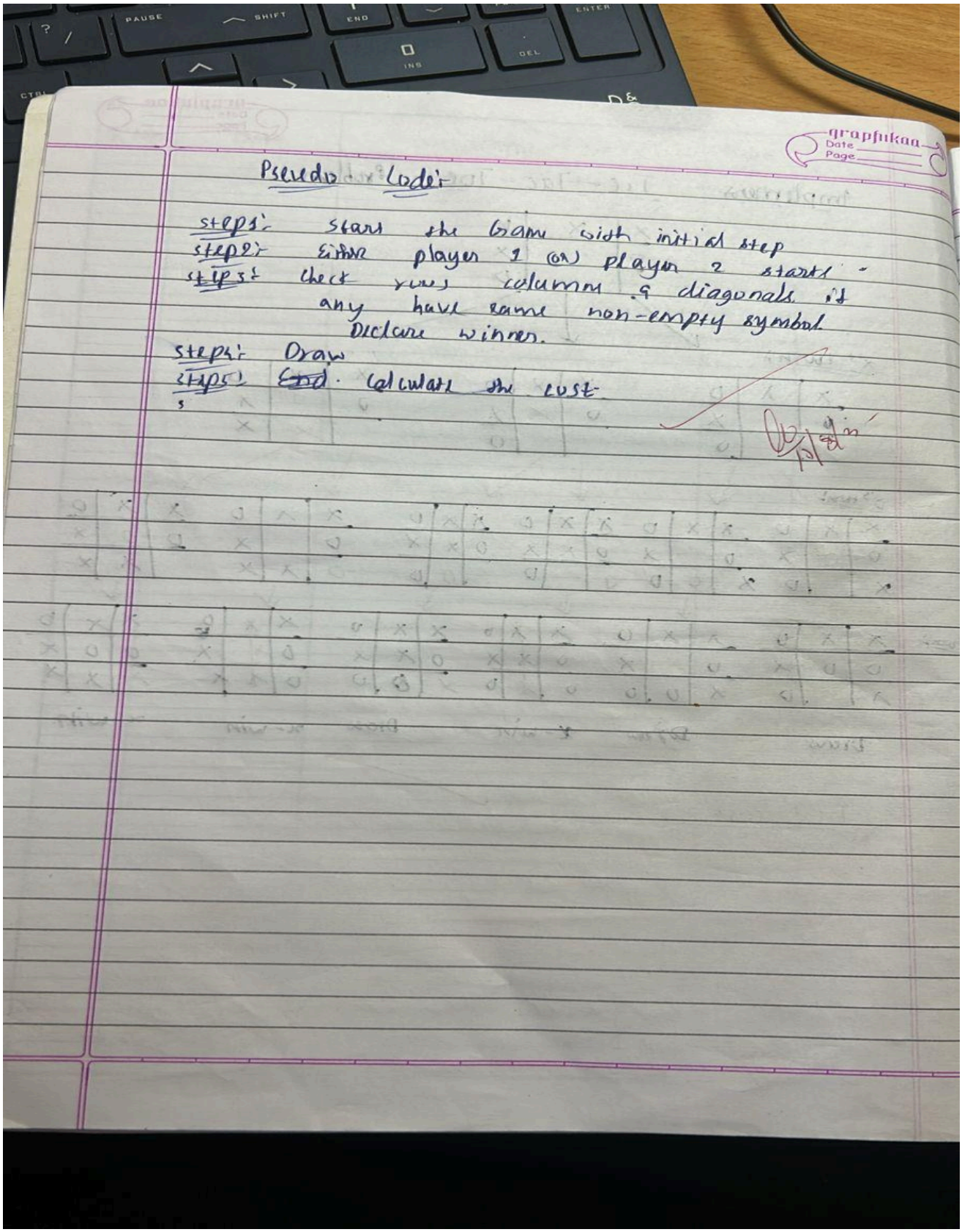
Program 1

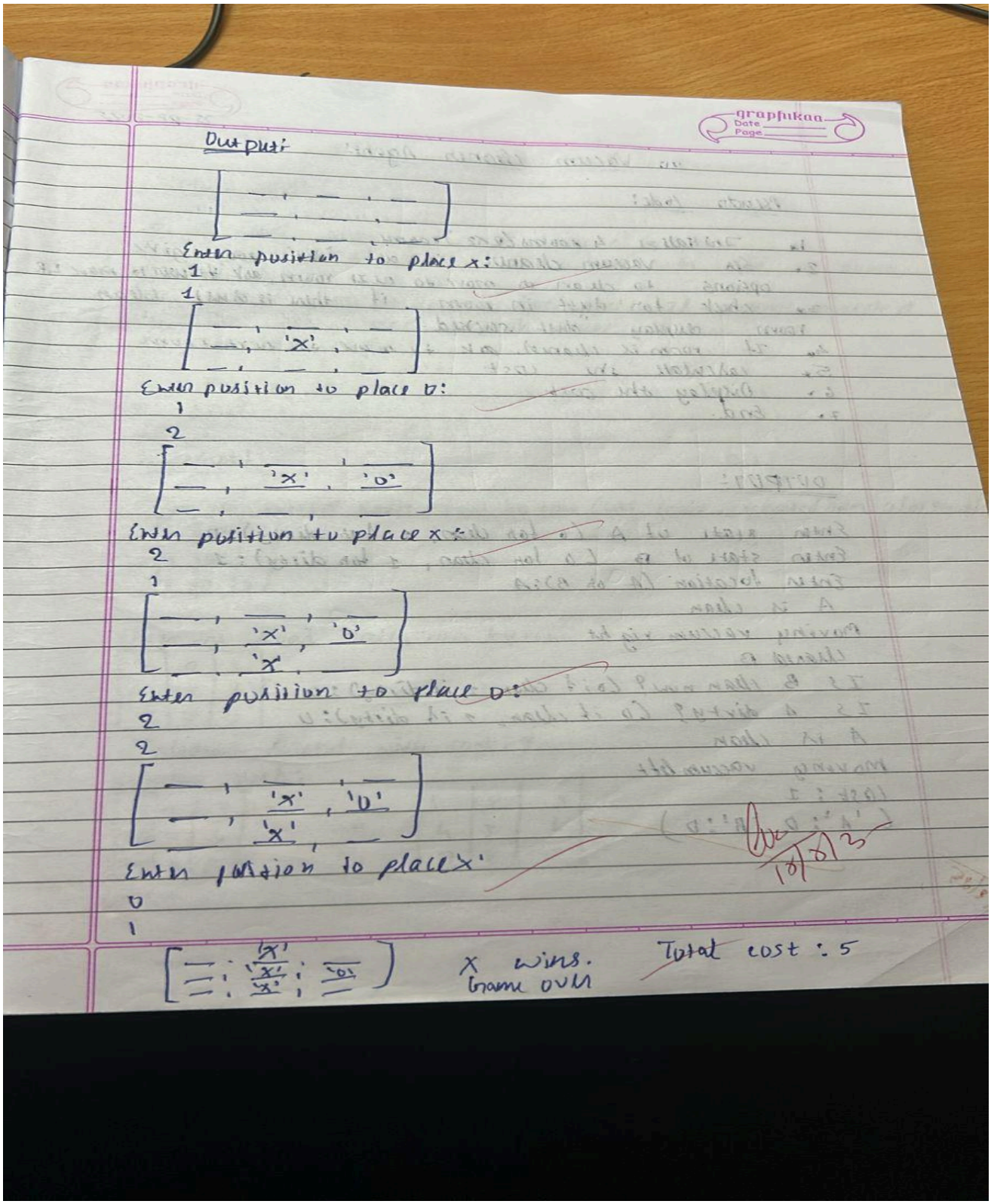
Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:




Pseudo Code:
step1: start the Game with initial step
step2: either player 1 (or) player 2 starts.
step3: check rows, columns & diagonals. if any have same non-empty symbol. Declare winner.
step4: Draw
step5: End. Calculate the cost.



Code:

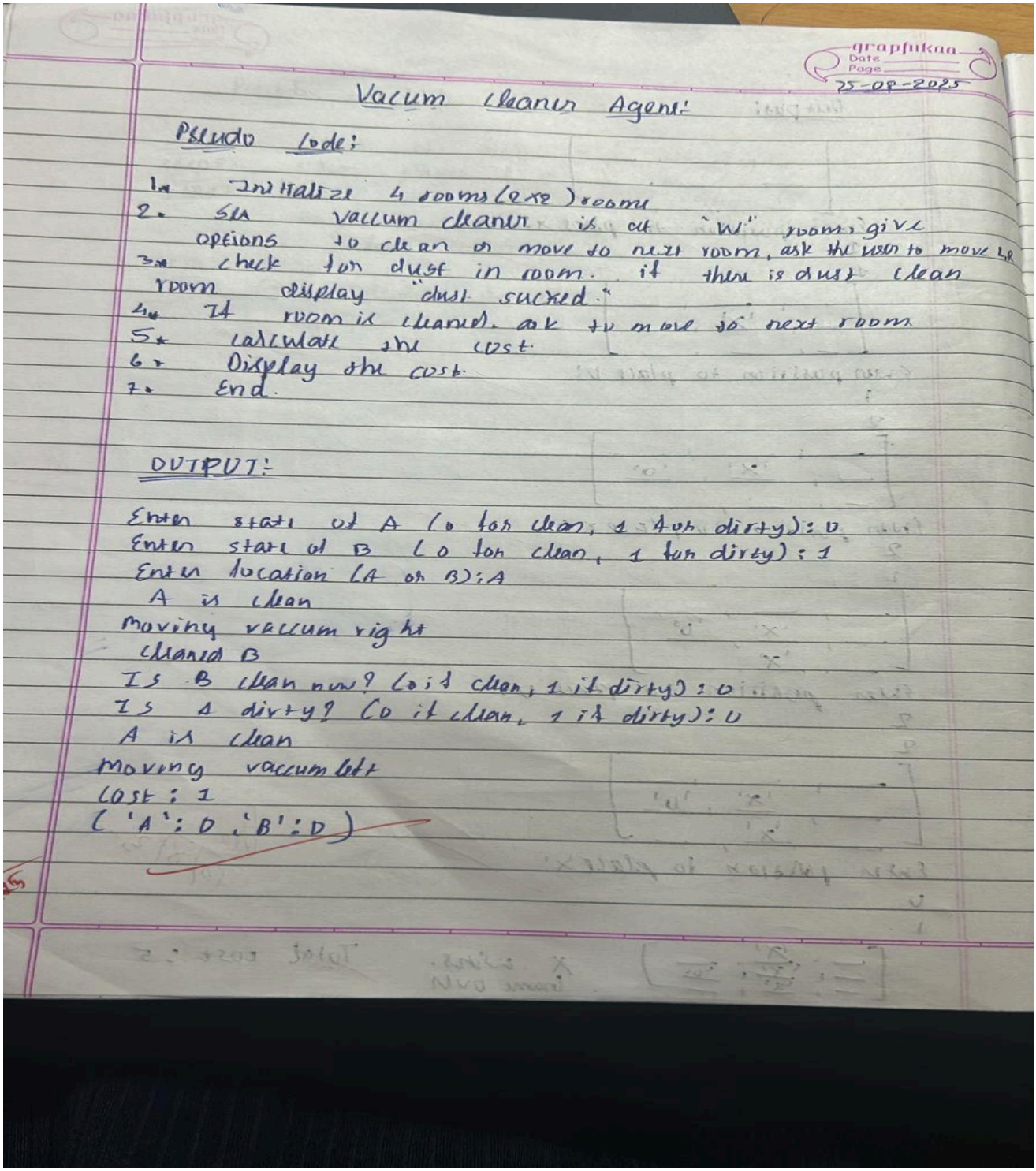
Implement Tic –Tac –Toe Game

code

```
def print_board(board):
    for row in board:
        print("|".join(row))
        print("-" * 5)

def check_win(board, player):
    for row in board:
        if all([cell == player for cell in row]):
            return True
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True
    if board[0][0] == board[1][1] == board[2][2] == player:
        return True
    if board[0][2] == board[1][1] == board[2][0] == player:
        return True
    return False

def check_draw(board):
    for row in board:
        if " " in row:
            return False
    return True

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    players = ["X", "O"]
    current_player = 0

    while True:
        print_board(board)
        player = players[current_player]
        try:
            row = int(input(f"Player {player}, enter row (0-2): "))
            col = int(input(f"Player {player}, enter column (0-2): "))
            if not (0 <= row <= 2 and 0 <= col <= 2):
                print("Invalid input. Row and column must be between 0 and 2.")
                continue
        except ValueError:
            print("Invalid input. Please enter a number.")
            continue


        if board[row][col] == " ":
            board[row][col] = player
            if check_win(board, player):
                print_board(board)
                print(f"Player {player} wins!")
                break
            elif check_draw(board):
                print_board(board)
                print("It's a draw!")
                break
            else:
                current_player = (current_player + 1) % 2
        else:
            print("That spot is already taken! Try again.")
```

```
play_game()
```

Implement vacuum cleaner agent



Vacuum cleaner Agent:

Pseudo code:

1. Initialize 4 rooms (2x2) rooms
2. Set vaccum cleaner is at "W" rooms, give options to clean or move to next room, ask the user to move L,R
3. check for dust in room. if there is dust clean room display "dust sucked."
4. If room is cleaned, ask to move to next room
5. calculate the cost
6. Display the cost.
7. End.

OUTPUT:

Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vaccum right
cleaned B
Is B clean now? (0: if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
A is clean
Moving vaccum left
Cost: 1
('A': 0, 'B': 0)

Code:

```
def vacuum_cleaner_agent():
    state_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    state_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    cost = 0
    states = {'A': state_A, 'B': state_B}

    if states['A'] == 0 and states['B'] == 0:
        print("Turning vacuum off")
        print(f"Cost: {cost}")
        print(states)
        return

    def clean_location(loc):
        nonlocal cost
        if states[loc] == 1:
            print(f"Cleaned {loc}.")
            states[loc] = 0
            cost += 1

    if location == 'A':
        if states['A'] == 1:
            clean_location('A')
        else:
            print("A is clean")
        print("Moving vacuum right")
        cost += 1
        if states['B'] == 1:
            clean_location('B')
        else:
            print("B is clean")

        print("Is B clean now? (0 if clean, 1 if dirty):", states['B'])
        print("Is A dirty? (0 if clean, 1 if dirty):", states['A'])

        if states['A'] == 1:
            print("Moving vacuum left")
            cost += 1
            clean_location('A')

    elif location == 'B':
        if states['B'] == 1:
            clean_location('B')
        else:
            print("B is clean")
        print("Moving vacuum left")
        cost += 1
        if states['A'] == 1:
            clean_location('A')
        else:
            print("A is clean")

        print("Is A clean now? (0 if clean, 1 if dirty):", states['A'])
        print("Is B dirty? (0 if clean, 1 if dirty):", states['B'])

        if states['B'] == 1:
```

```
            print("Moving vacuum right")
            cost += 1
            clean_location('B')

    print(f"Cost: {cost}")
    print(states)

vacuum_cleaner_agent()
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

grapfikan
Date
Page

BFS without Heuristic Approach:

Pseudo code:

1. Initial state and goal state given
2. from initial state will have 4 depth up, down, and right
3. Then check with the goal state. If doesnot match do.
4. Do the next branch state.
5. If matched
6. Calculate the cost
7. Display the cost
8. End.

Output:

Enter the initial state, row by row (use space-seperated num, 0 for empty)

1	2	3
4	5	6
7	0	8

Enter the goal state, row by row

1	2	3
4	5	6
7	8	0

solution found with cost: 1

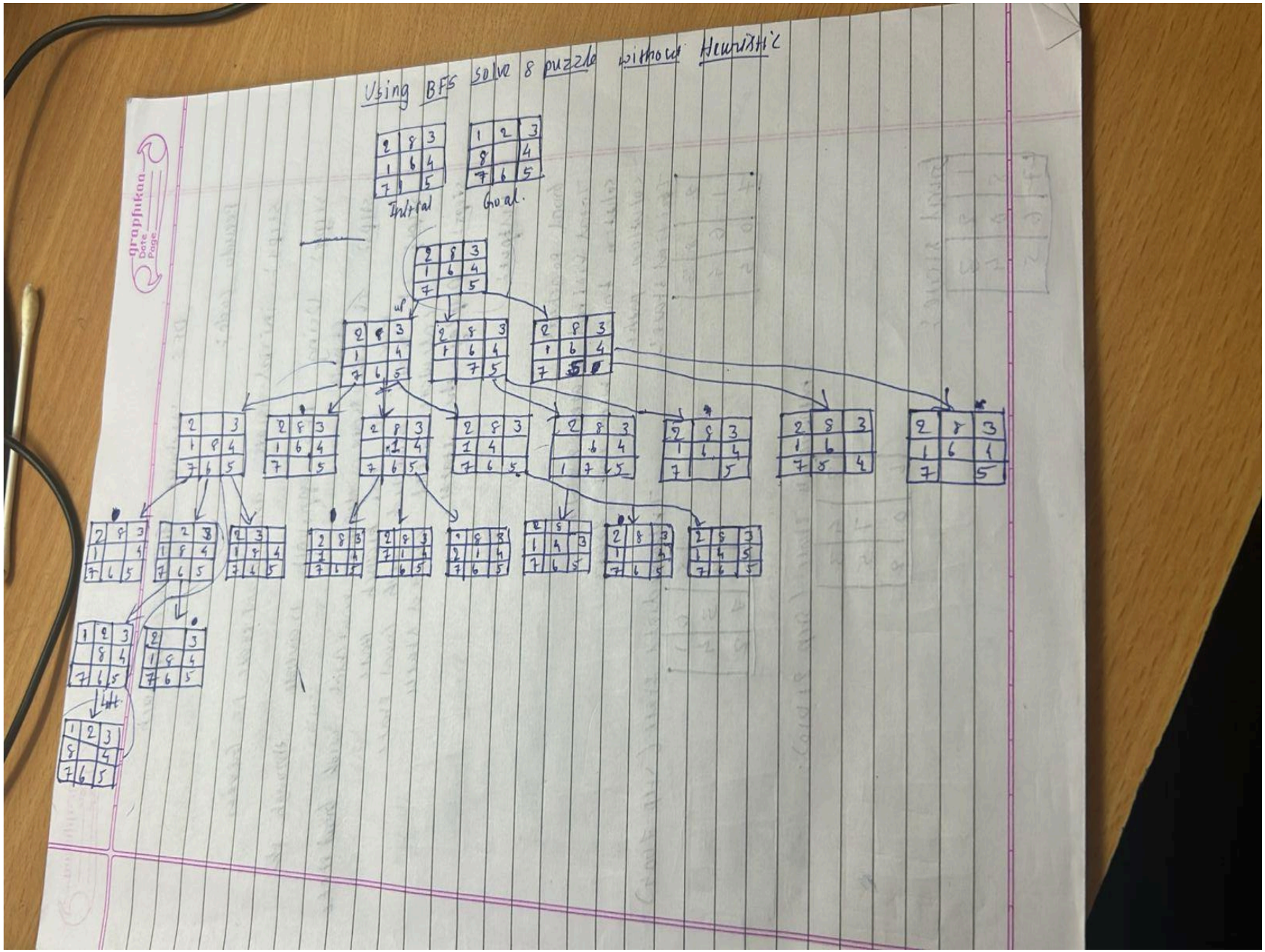
solution path:

1	2	3
4	5	6
7	0	8

→

1	2	3
4	5	6
7	8	0

25/08/25


Using BFS solve 8 puzzle without Heuristic
Initial
Goal.

Code

```
from collections import deque


initial_state = ((2, 8, 3),
                 (1, 6, 4),
                 (7, 0, 5))

goal_state = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

directions = {'Up': (-1, 0), 'Down': (1, 0), 'Left': (0, -1), 'Right': (0, 1)}

def get_blank_pos(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    r1, c1 = pos1
    r2, c2 = pos2
    state_list[r1][c1], state_list[r2][c2] = state_list[r2][c2],
state_list[r1][c1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    r, c = get_blank_pos(state)
    for move, (dr, dc) in directions.items():
        nr, nc = r + dr, c + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            new_state = swap_positions(state, (r, c), (nr, nc))
            neighbors.append((new_state, move))
    return neighbors

def bfs_8_puzzle(start, goal):
    queue = deque()
    visited = set()
    visited.add(start)
    queue.append((start, []))

    levels = []

    while queue:
        level_size = len(queue)
        current_level_states = []

        for _ in range(level_size):
            state, path = queue.popleft()
            current_level_states.append(state)
            if state == goal:
                return path, levels + [current_level_states], len(visited)

            for neighbor, move in get_neighbors(state):
```

```
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, path + [move]))

        levels.append(current_level_states)

    return None, levels, len(visited)

solution_path, level_states, total_visited = bfs_8_puzzle(initial_state,
goal_state)

print(f"Solution length: {len(solution_path)} moves")
print("Solution moves:", solution_path)
print(f"Total states visited: {total_visited}\n")

print("States level-wise:")
for i, level in enumerate(level_states):
    print(f"\nLevel {i}:")
    for state in level:
        for row in state:
            print(row)
        print("---")
```

program

Implement Iterative deepening search algorithm

graphikaa
Date
Page

Iterative deepning:

ALGORITHM:

function ITERATIVE-DEEPENING-SEARCH
inputs: problem, a problem
for depth ← 0 to ∞ do
result ← DEPTH-LIMITED-SEARCH
if result ≠ cutoff then return result
end

output:

initial state:

2	8	3
1	6	4
7	0	5

final state:

1	2	3
8	0	4
7	6	5

No solution at depth 0:
No solution at depth 1:
:
found solution at depth 5:

Total visited states: 56
Solution found with cost: 5.

code

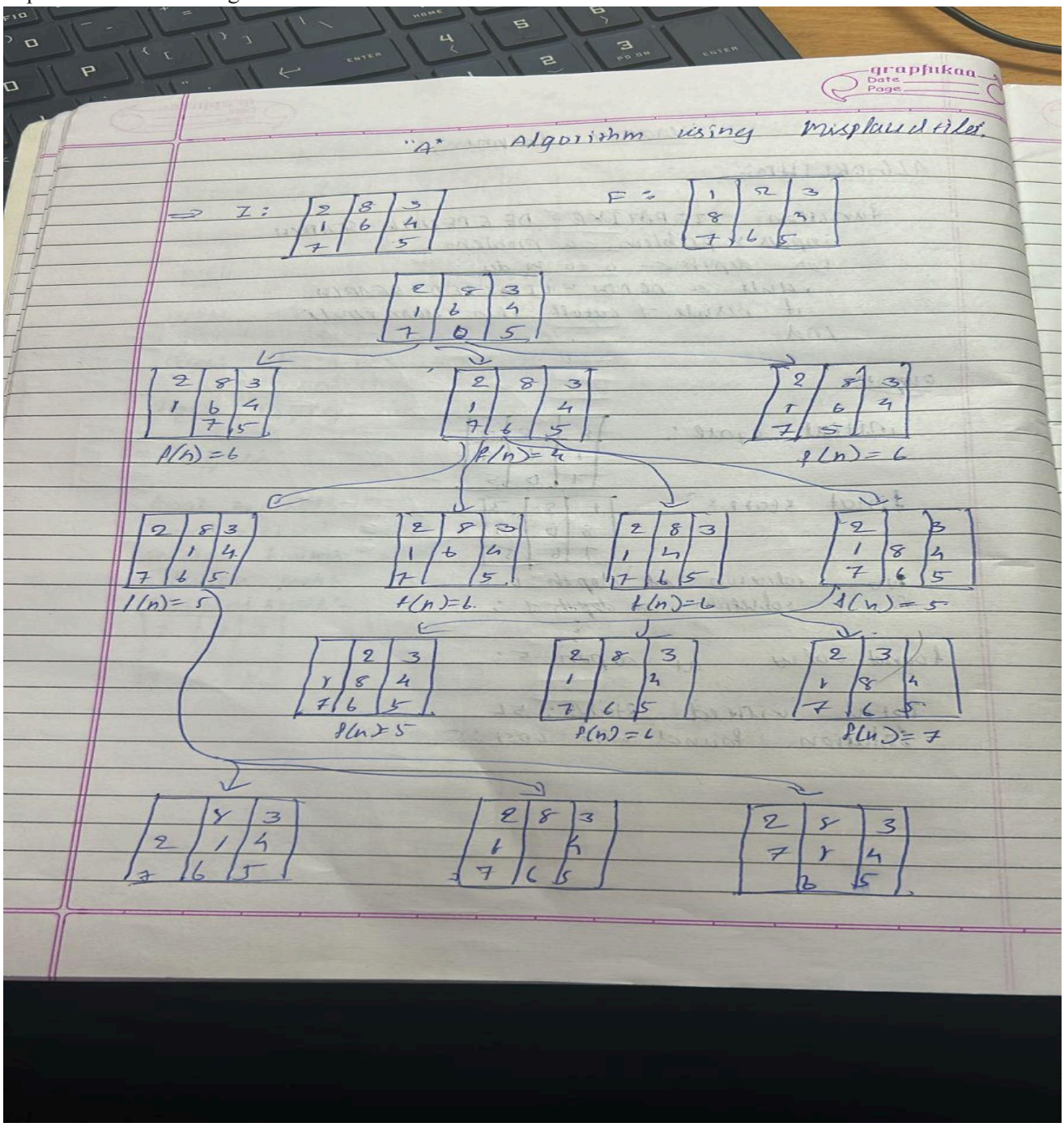
```
from collections import deque


initial_state = ((2, 8, 3),
                 (1, 6, 4),
                 (7, 0, 5))

goal_state = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

directions = {'Up': (-1, 0), 'Down': (1, 0), 'Left': (0, -1), 'Right':
(0, 1)}

def get_blank_pos(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    r1, c1 = pos1
    r2, c2 = pos2
    state_list[r1][c1], state_list[r2][c2] = state_list[r2][c2],
state_list[r1][c1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    r, c = get_blank_pos(state)
    for move, (dr, dc) in directions.items():
        nr, nc = r + dr, c + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            new_state = swap_positions(state, (r, c), (nr, nc))
            neighbors.append((new_state, move))
    return neighbors

def bfs_8_puzzle(start, goal):
    queue = deque()
    visited = set()
    visited.add(start)
    queue.append((start, []))

    levels = []

    while queue:
        level_size = len(queue)
        current_level_states = []

        for _ in range(level_size):
```

```
            state, path = queue.popleft()
            current_level_states.append(state)
            if state == goal:
                return path, levels + [current_level_states],
len(visited)

            for neighbor, move in get_neighbors(state):
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, path + [move]))

        levels.append(current_level_states)

    return None, levels, len(visited)

solution_path, level_states, total_visited = bfs_8_puzzle(initial_state,
goal_state)

print(f"Solution length: {len(solution_path)} moves")
print("Solution moves:", solution_path)
print(f"Total states visited: {total_visited}\n")

print("States level-wise:")
for i, level in enumerate(level_states):
    print(f"\nLevel {i}:")
    for state in level:
        for row in state:
            print(row)
        print("---")
```

```
Solution length: 5 moves
Solution moves: ['Up', 'Up', 'Left', 'Down', 'Right']
Total states visited: 62
```

program 3:

Implement A* search algorithm



output

start state : 2 8 3 1 6 4 7 _ 5

Goal state : 1 2 3 8 _ 4 7 6 5

solving puzzle :

solution found at 5th depth

path :

2	8	3	→	2	8	3	→	b	2	3
1	b	4		1	0	4		1	8	4
7	0	5		7	b	5		7	b	5

↓

1	2	3	←	1	2	3		
8	0	4		b	8	4		
7	b	5		7	b	5		

Total cost is '5'

⇒ "A* Algorithm using Manhatten Distance"

I: | 1 | 5 | 8 |
| 3 | 2 | |
| 4 | 6 | 7 |

F: | 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

1	5	8
3	2	
4	6	7

g(n) = 0

1	5	8
3		2
4	6	7

f(n) = 15
m(n) = 0+2+3+1+1
+2+2+3=14

1	5	
3	2	8
4	6	7

f(n) = 13
m(n) = 0+1+3+1
+1+2+2+2

1	5	8
3	2	7
4	6	

f(n) = 15
m(n) = 0+1+3+1+
2+3+3.

1		5
3	2	8
4	6	7

f(n) = 15
m(n) = 0+1+3+1+2+2+2
+2+2.

1	2	5
3		8
4	6	7

f(n) = 15
m(n) = 0+1+3+1+1+2+2+3

g = 2

	1	5
3	2	8
4	6	7

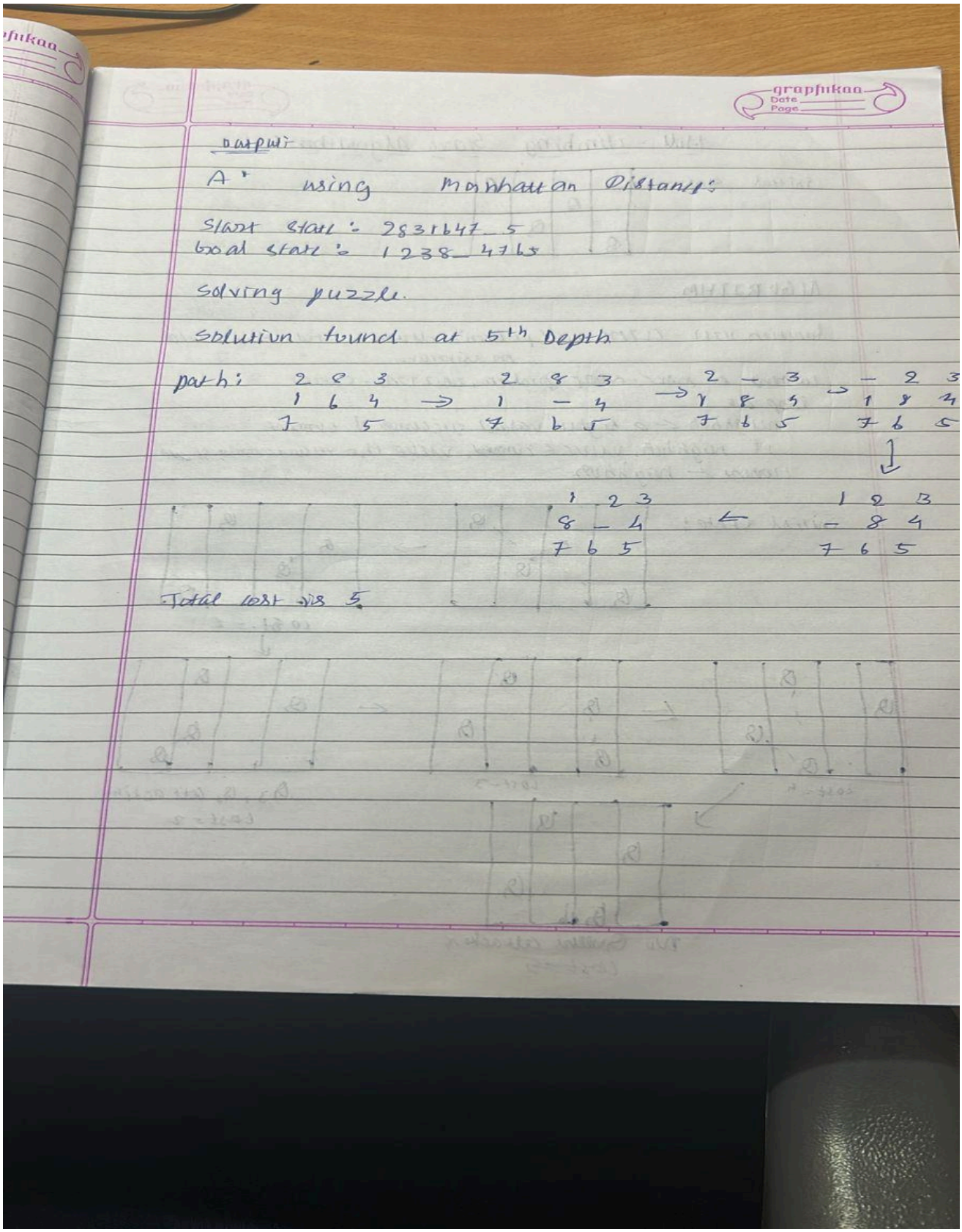
m(n) = 1+1+3+1+2
+2+2+2

1	2	5
3		8
4	6	7

m(n) = 0+0+3+1+2
+2+2+2

2	5	
3	2	8
4	6	7

X



code:

```
import heapq
```

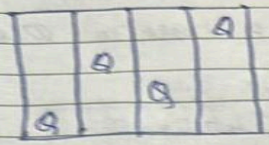
```
def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                value = state[i][j]
                # Find the position of the value in the goal
state
                for gi in range(3):
                    for gj in range(3):
                        if goal[gi][gj] == value:
                            goal_pos = (gi, gj)
                            break
                    else:
                        continue
                    break
                distance += abs(i - goal_pos[0]) + abs(j -
goal_pos[1])
    return distance

def get_neighbors(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break
        else:
            continue
        break

    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] =
new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in
new_state))
    return neighbors

def astar_search_manhattan(initial, goal):
    frontier = [(manhattan_distance(initial, goal), 0, initial)]
    explored = set()
    parent = {}
    cost = {initial: 0}

    while frontier:
        f, g, current = heapq.heappop(frontier)

        if current == goal:
            path = []
            while current in parent:
                path.append(current)
                current = parent[current]
```

```
            path.append(initial)
            return path[::-1]

        explored.add(current)

        for neighbor in get_neighbors(current):
            new_cost = cost[current] + 1
            if neighbor not in cost or new_cost < cost[neighbor]:
                cost[neighbor] = new_cost
                priority = new_cost +
manhattan_distance(neighbor, goal)
                heapq.heappush(frontier, (priority, new_cost,
neighbor))
                parent[neighbor] = current
    return None

def get_state_input(prompt):
    print(prompt)
    state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        state.append(row)
    return tuple(tuple(row) for row in state)

initial_state_m = get_state_input("Enter the initial state for
Manhattan distance (3 rows of 3 numbers separated by spaces, use
0 for the blank):")
goal_state_m = get_state_input("Enter the goal state for
Manhattan distance (3 rows of 3 numbers separated by spaces, use
0 for the blank):")

path_m = astar_search_manhattan(initial_state_m, goal_state_m)

if path_m:
    print("Solution found using Manhattan distance:")
    for step in path_m:
        for row in step:
            print(row)
        print()
else:
    print("No solution found using Manhattan distance.")
```

program 4: Implement Hill Climbing search algorithm to solve N-Queens problem

Hill - climbing Search Algorithm:

initial state:



ALGORITHM

function HILL-CLIMBING(problem) returns a state that is local minimum

current ← MAKE-NODE(problem.INITIAL-STATE)

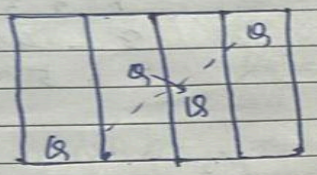
loop do

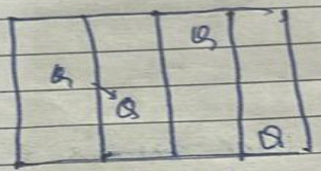
neighbor ← a highest valued successor of current

if neighbor.VALUE ≤ current.VALUE then return current.STATE

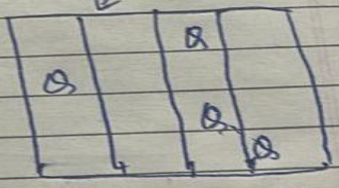
current ← neighbor

initial state:



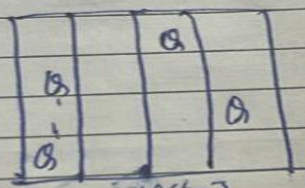


cost = 1

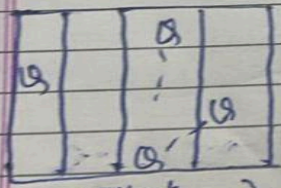


Q_3, Q_4 attacking

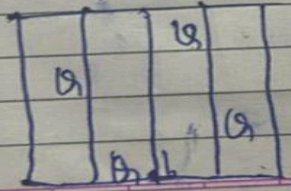
cost = 2



cost = 3



cost = 4



No Queens attacking

cost = 5

OUTPUT:

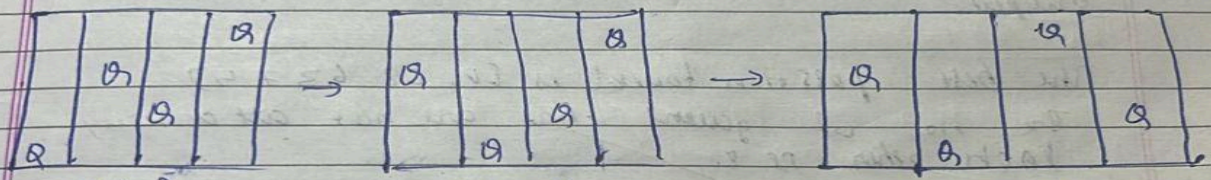
Enter the initial positions of the 4 queens
column 0 : 3
column 1 : 1
column 2 : 2
column 3 : 0

Start state:

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

cost = 2

final path:

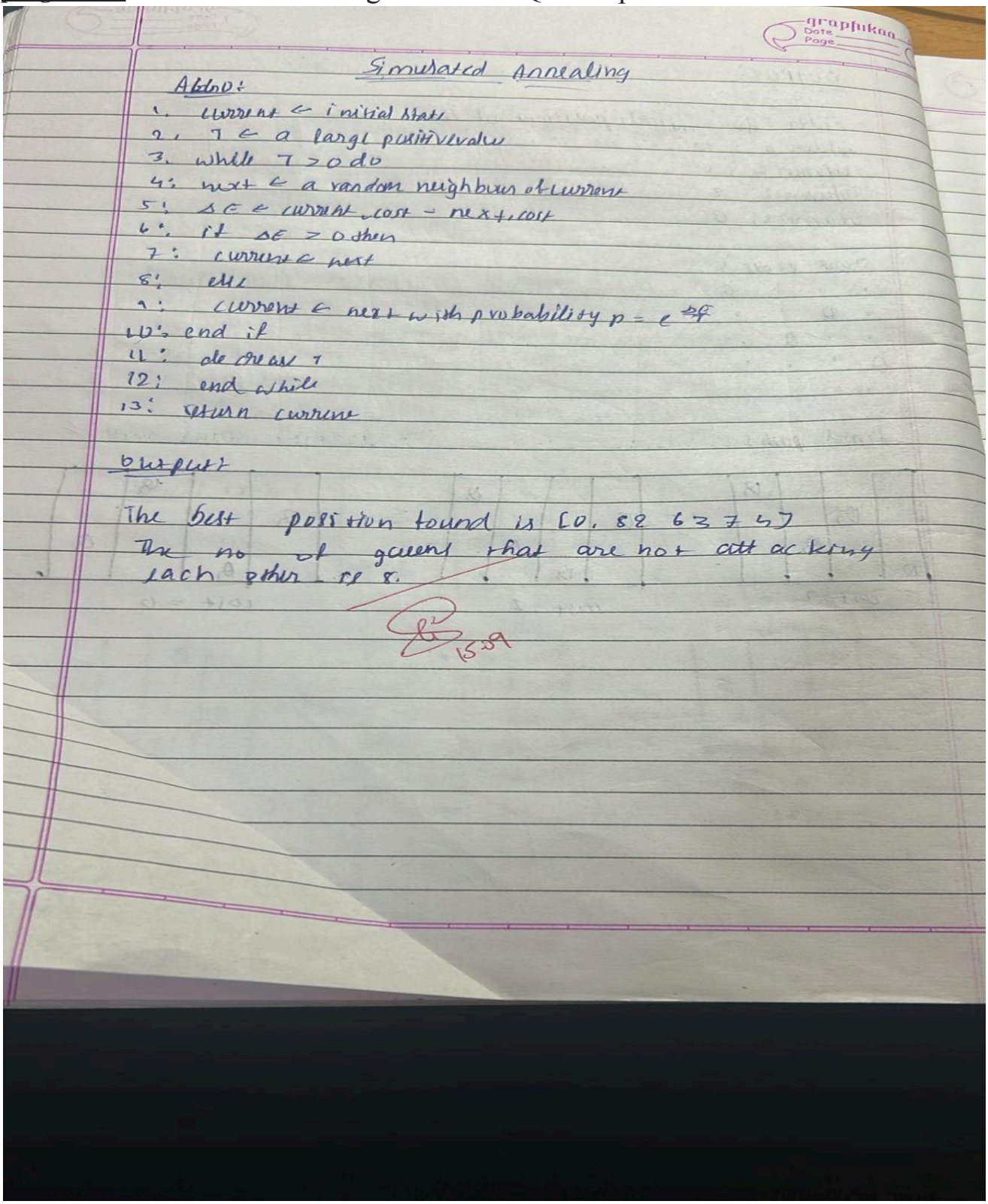


code:

```
import random

def print_board(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 heuristic(state):
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:
                attacks += 1
            if abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                neighbor = list(state)
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors

def hill_climbing(n=4, initial_state=None):
    if initial_state is None:
        current = [random.randint(0, n-1) for _ in range(n)]
    else:
        current = initial_state
    current_heuristic = heuristic(current)
    print(f"Initial state: {current} with heuristic cost:
{current_heuristic}")
    print("Initial board:")
    print_board(current)

    while True:
        neighbors = get_neighbors(current)
        neighbor_heuristics = [(heuristic(nei), nei) for nei in
neighbors]
        best_heuristic, best_neighbor = min(neighbor_heuristics,
key=lambda x: x[0])

        if best_heuristic >= current_heuristic:
            break
```

```
        print(f"\nCurrent state: {current} with heuristic cost:
{current_heuristic}")
        print("Current board:")
        print_board(current)

        current, current_heuristic = best_neighbor,
best_heuristic

    print(f"\nFinal state: {current} with heuristic cost:
{current_heuristic}")
    print("Final board:")
    print_board(current)

    return current, current_heuristic

def get_user_input():
    n = int(input("Enter the size of the board (default is 4): ")
or 4)
    initial_state = input(f"Enter the initial state as a
space-separated list of row indices for each column (length {n}):
")
    initial_state = list(map(int, initial_state.split()))
    return n, initial_state

n, initial_state = get_user_input()
solution, conflicts = hill_climbing(n, initial_state)
print("\nFinal board with conflicts =", conflicts)
```

program 5: Simulated Annealing to Solve 8-Queens problem



Simulated Annealing

Algo:

1. current ← initial state
2. T ← a large positive value
3. while T > 0 do
4. next ← a random neighbour of current
5. ΔE ← current.cost − next.cost
6. if ΔE > 0 then
7. current ← next
8. else
9. current ← next with probability $p = e^{\frac{\Delta E}{T}}$
10. end if
11. decrease T
12. end while
13. return current

Output:

The best position found is [0. 5 2 6 3 7 4 1]
The no of queens that are not attacking each other is 8.

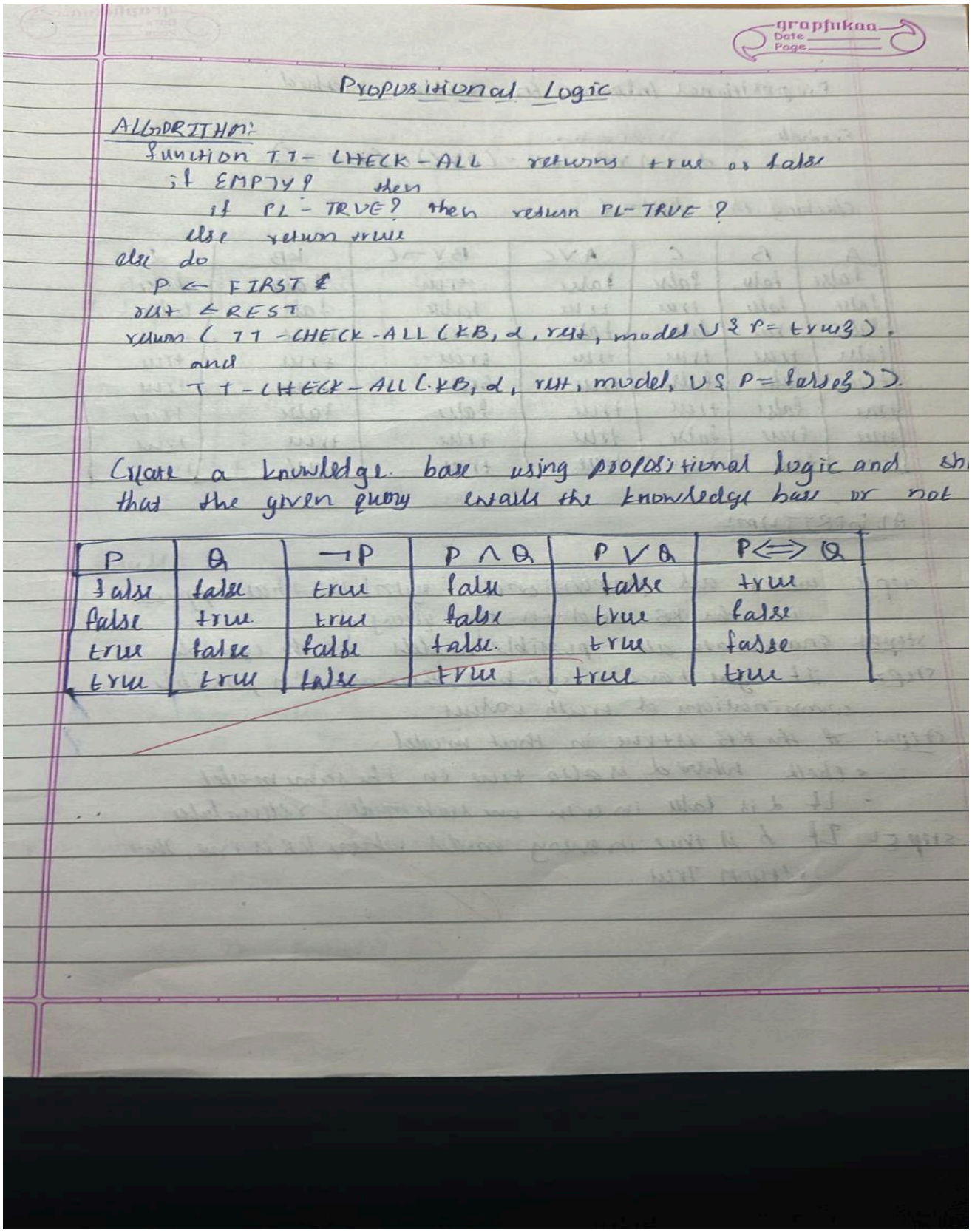
15/9

code:

```
import math
import random

# Example objective function (you can replace this with any
function)
# Let's say we want to minimize f(x) = x^2 + 10*sin(x)
def objective_function(x):
    return x**2 + 10 * math.sin(x)

# Simulated Annealing
def simulated_annealing(objective, bounds, max_iterations,
initial_temp, cooling_rate):
    # Random initial solution
    current = random.uniform(bounds[0], bounds[1])
    current_energy = objective(current)

    # Best found solution
    best = current
    best_energy = current_energy

    temp = initial_temp

    for i in range(max_iterations):
        # New candidate solution (neighbor)
        candidate = current + random.uniform(-1, 1)  # small
random move
        candidate = max(min(candidate, bounds[1]), bounds[0])  #
keep inside bounds
        candidate_energy = objective(candidate)

        # Energy difference
        delta_e = candidate_energy - current_energy

        # Accept candidate if better OR with probability
exp(-ΔE/T)
        if delta_e < 0 or random.random() < math.exp(-delta_e /
temp):
            current, current_energy = candidate, candidate_energy

            # Check for new best
            if current_energy < best_energy:
                best, best_energy = current, current_energy

        # Decrease temperature
        temp = temp * cooling_rate

        # Debug: print iteration progress
        # print(f"Iter {i}, Temp={temp:.4f}, Best={best:.4f},
Energy={best_energy:.4f}")

    return best, best_energy

# ------------------------------
# Example Run
# ------------------------------

bounds = [-10, 10]          # search space for x
```

```
max_iterations = 1000       # number of iterations
initial_temp = 100.0        # starting temperature
cooling_rate = 0.99         # cooling rate (0.8 - 0.99 is
typical)

best_solution, best_value =
simulated_annealing(objective_function, bounds,
                                              max_iterations,
initial_temp, cooling_rate)

print("Best solution found: x =", best_solution)
print("Objective function value:", best_value)
```

program 6: Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.



Propositional Logic

ALGORITHM:

function TT-CHECK-ALL returns true or false
  if EMPTY? then
    if PL-TRUE? then return PL-TRUE?
    else return true
  else do
    P ← FIRST
    rest ← REST
    return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true}) and TT-CHECK-ALL(KB, α, rest, model ∪ {P = false}))

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

P	Q	¬P	P ∧ Q	P ∨ Q	P ⟺ Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

code:

```
import itertools
```

```
import pandas as pd
import re

def replace_implications(expr):
    """
    Replace every X => Y with (not X or Y).
    This uses regex with a callback to avoid partial string overwrites.
    """
    # Pattern: capture left side and right side around =>
    # Made more flexible to handle various expressions
    pattern = r'([^=><]+?)\s*=>\s*([^=><]+?)(?=\s|$|[&|)])'
    while re.search(pattern, expr):
        expr = re.sub(pattern,
                      lambda m: f"(not {m.group(1).strip()} or
{m.group(2).strip()})",
                      expr,
                      count=1)
    return expr

def pl_true(sentence, model):
    expr = sentence.strip()
    expr = expr.replace("<=>", "==")
    expr = replace_implications(expr)

    # Replace propositional symbols with their truth values safely
    for sym, val in model.items():
        expr = re.sub(rf'\b{sym}\b', str(val), expr)

    # Clean up spacing and add proper spacing for boolean operators
    expr = re.sub(r'\s+', ' ', expr)  # Remove extra spaces
    expr = expr.replace(" and ", " and ").replace(" or ", " or ").replace("
not ", " not ")

    return eval(expr)

def get_symbols(KB, alpha):
    symbols = set()
    for sentence in KB + [alpha]:
        # Find all alphabetic tokens (propositional variables)
        for token in re.findall(r'\b[A-Za-z]+\b', sentence):
            if token not in ['and', 'or', 'not']:  # Exclude boolean
operators
                symbols.add(token)
    return sorted(list(symbols))

def tt_entails(KB, alpha):
    symbols = get_symbols(KB, alpha)
    rows = []
    entails = True

    for values in itertools.product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))

        try:
            kb_val = all(pl_true(sentence, model) for sentence in KB)
            alpha_val = pl_true(alpha, model)

            rows.append({**model, "KB": kb_val, "alpha": alpha_val})
```

```
            if kb_val and not alpha_val:
                entails = False
        except Exception as e:
            print(f"Error evaluating with model {model}: {e}")
            return False

    df = pd.DataFrame(rows)

    # Create a beautiful formatted table
    print("\n" + "="*50)
    print("                TRUTH TABLE")
    print("="*50)

    # Get column widths for proper alignment
    col_widths = {}
    for col in df.columns:
        col_widths[col] = max(len(str(col)),
df[col].astype(str).str.len().max())

    # Calculate total table width
    table_width = sum(col_widths.values()) + len(df.columns) * 3 - 1

    # Print top border
    print("┌" + "─" * table_width + "┐")

    # Print header
    header = "│"
    for col in df.columns:
        header += f" {col:^{col_widths[col]}} │"
    print(header)

    # Print separator
    separator = "├"
    for col in df.columns:
        separator += "─" * (col_widths[col] + 2) + "┼"
    separator = separator[:-1] + "┤"
    print(separator)

    # Print rows
    for _, row in df.iterrows():
        row_str = "│"
        for col in df.columns:
            value = str(row[col])
            row_str += f" {value:^{col_widths[col]}} │"
        print(row_str)

    # Print bottom border
    print("└" + "─" * table_width + "┘")

    # Print result with styling
    print("\n" + "="*50)
    result_text = f"KB ENTAILS ALPHA: {'✓ YES' if entails else '✗ NO'}"
    print(f"{result_text:^50}")
    print("="*50)
    return entails

# --- Interactive input ---
```

```
print("Enter Knowledge Base (KB) sentences, separated by commas.")
print("Use symbols like A, B, C and operators: and, or, not, =>, <=>")
kb_input = input("KB: ").strip()
KB = [x.strip() for x in kb_input.split(",")]
alpha = input("Enter query (alpha): ").strip()
result = tt_entails(KB, alpha)
print(f"Result: {result}")
```

program 7: Implement unification in first order logic

Output

Enter knowledge Base (use, &, |, ~ for AND, OR, NOT):
(A|B) & (B|~C)
Enter Query: A|B

Truth Table:

A	B	C	KB	Query
F	F	F	0	False
F	F	T	0	False
F	T	F	1	True
F	T	T	1	True
T	F	F	1	True
T	F	T	0	True
T	T	F	1	True
T	T	T	1	True

Result:
KB entails Query (True in all cases).

---

(3)

Consider $S \in T$ as variables and following relations :→

a : $\neg(S \vee T)$

b : $(S \wedge T)$

c : $T \vee \neg T$

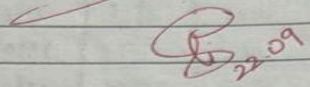
write Truth Table and show whether

i) a entails b

ii) a entails c.

S	T	a (¬(S ∨ T))	b (S ∧ T)	c (¬T ∨ ¬T)
T	T	F	T	F
T	F	F	F	T
F	T	F	F	T
F	F	T	F	T

i) a entails b is false
ii) a ~~entails~~ c is true

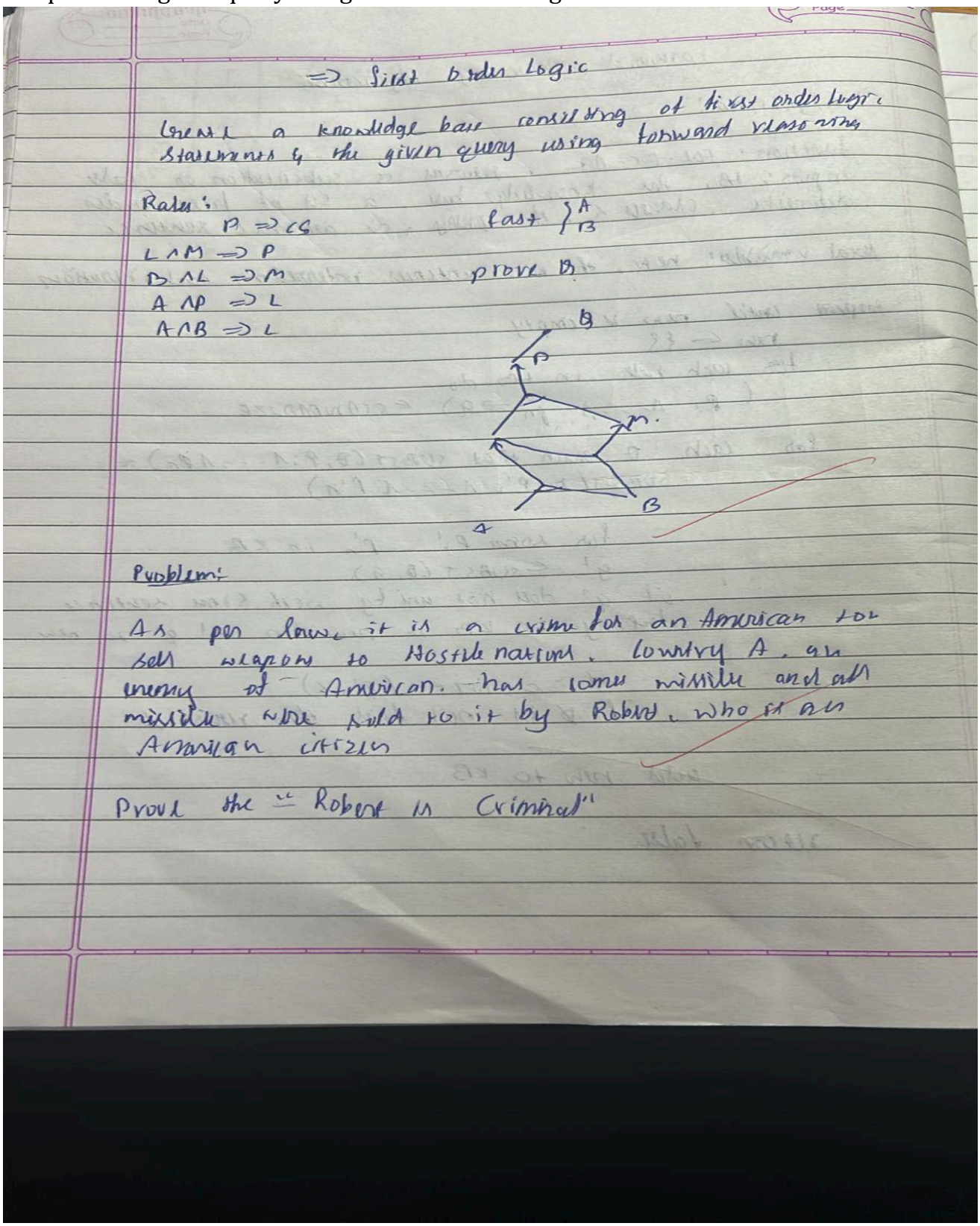


Code

```
def is_variable(term):
    return isinstance(term, str) and term[0].islower() and term.isalpha()

def is_compound(term):
    return isinstance(term, tuple)

def occur_check(var, term, subst):
    if var == term:
        return True
    elif is_variable(term) and term in subst:
        return occur_check(var, subst[term], subst)
    elif is_compound(term):
        return any(occur_check(var, t, subst) for t in term[1])
    else:
        return False

def substitute(term, subst):
    if is_variable(term):
        return substitute(subst[term], subst) if term in subst else term
    elif is_compound(term):
        return (term[0], [substitute(t, subst) for t in term[1]])
    else:
        return term

def unify(x, y, subst=None):
    if subst is None:
        subst = {}

    x = substitute(x, subst)
    y = substitute(y, subst)

    if x == y:
        return subst

    elif is_variable(x):
        if occur_check(x, y, subst):
            raise Exception(f"Occurs check failed: {x} in {y}")
        subst[x] = y
        return subst

    elif is_variable(y):
        if occur_check(y, x, subst):
            raise Exception(f"Occurs check failed: {y} in {x}")
        subst[y] = x
        return subst

    elif is_compound(x) and is_compound(y):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            raise Exception(f"Function symbols or arity mismatch: {x[0]} vs
{y[0]}")
        for a, b in zip(x[1], y[1]):
            subst = unify(a, b, subst)
        return subst

    else:
        raise Exception(f"Cannot unify {x} and {y}")
```

```
# -------------------- Example --------------------
if __name__ == "__main__":
    # Example 1: Unify P(x, y) and P(a, b)
    expr1 = ('P', ['x', 'y'])
    expr2 = ('P', ['a', 'b'])

    try:
        result = unify(expr1, expr2)
        print("Substitution:", result)
    except Exception as e:
        print("Unification failed:", e)

    # Example 2: Unify Q(x, f(y)) and Q(f(a), f(b))
    expr3 = ('Q', ['x', ('f', ['y'])])
    expr4 = ('Q', [('f', ['a']), ('f', ['b'])])

    try:
        result = unify(expr3, expr4)
        print("Substitution:", result)
    except Exception as e:
        print("Unification failed:", e)
```

program 8: Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.



=> First order Logic

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

Rules:

$P \Rightarrow CS$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

fact $\begin{cases} A \\ B \end{cases}$

prove B

Problem:

As per law, it is a crime for an American to sell weapons to Hostile nations. Country A, an enemy of American, has some missile and all missile were sold to it by Robert, who is an American citizen

Prove the "Robert is Criminal"

graphikaa
Date
Page

It say P, Q, and R are variable

American (P) ∧ weapon (Q) ∧ sells (P, Q, R) ∧ Hostile (R) ⇒ Criminal (P)

Country A has some missile

∃ x owns (A, x) ∧ missile (x)

or

owns (A, T1)

↓

Missile (T1)

∀ x missile (x) ∧ owns (A, x) ⇒ sells (Robert, x, A)

Missile (x) ⇒ Weapon (x)

∀ x Enemy (x, America) ⇒ Hostile (x)

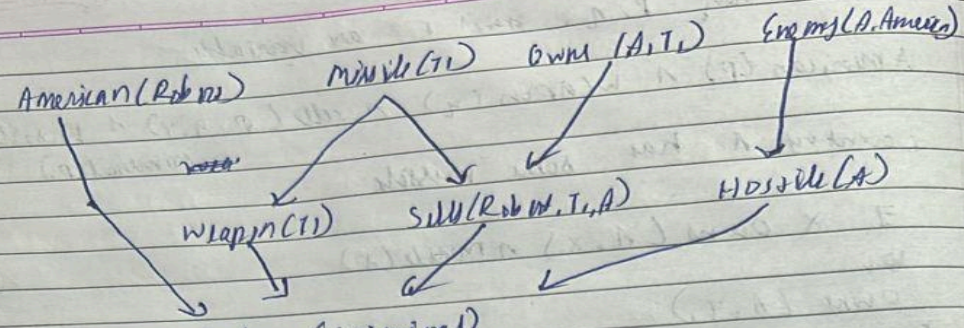
Robert is an American

American (Robert)

The country A, an enemy of America

Enemy (A, America)

∴ Robert is Criminal.



Output:

Step 1: Enemy (A, America) → Hostile (A)
Step 2: Missile (T1) → Weapon (T1)
Step 3: Missile (T1) ∧ Owns (A, T1) → Sells (Robert, T1, A)
Step 4: American (Robert) ∧ Weapon (T1) ∧ Sells (Robert, T1, A) ∧ Hostile (A) → Criminal (Robert)

∴ Robert is Criminal.

DW
27/10

code:

```
################################
class Term:
    """Base class for terms in first-order logic"""
    pass

class Constant(Term):
    """Represents a constant"""
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return isinstance(other, Constant) and self.name == other.name

    def __repr__(self):
        return self.name

    def __hash__(self):
        return hash(('Constant', self.name))

class Variable(Term):
    """Represents a variable"""
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return isinstance(other, Variable) and self.name == other.name

    def __repr__(self):
        return self.name

    def __hash__(self):
        return hash(('Variable', self.name))

class Predicate(Term):
    """Represents a predicate with arguments"""
    def __init__(self, name, args):
        self.name = name
        self.args = args if isinstance(args, list) else [args]

    def __eq__(self, other):
        return (isinstance(other, Predicate) and
                self.name == other.name and
                len(self.args) == len(other.args) and
                all(a == b for a, b in zip(self.args, other.args)))

    def __repr__(self):
        return f"{self.name}({', '.join(str(arg) for arg in self.args)})"

def occurs_check(var, term, subst):
    """Check if variable occurs in term (prevents infinite structures)"""
    if var == term:
        return True
    elif isinstance(term, Variable) and term in subst:
```

```
        return occurs_check(var, subst[term], subst)
    elif isinstance(term, Predicate):
        return any(occurs_check(var, arg, subst) for arg in term.args)
    return False

def apply_substitution(term, subst):
    """Apply substitution to a term"""
    if isinstance(term, Variable):
        if term in subst:
            return apply_substitution(subst[term], subst)
        return term
    elif isinstance(term, Predicate):
        new_args = [apply_substitution(arg, subst) for arg in term.args]
        return Predicate(term.name, new_args)
    else:
        return term

def unify(term1, term2, subst=None):
    """
    Unification Algorithm
    Returns substitution set if unification succeeds, None if it fails
    """
    if subst is None:
        subst = {}

    # Apply existing substitutions
    term1 = apply_substitution(term1, subst)
    term2 = apply_substitution(term2, subst)

    # Step 1: If term1 or term2 is a variable or constant
    # Step 1a: If both are identical
    if term1 == term2:
        return subst

    # Step 1b: If term1 is a variable
    elif isinstance(term1, Variable):
        if occurs_check(term1, term2, subst):
            return None  # FAILURE
        else:
            new_subst = subst.copy()
            new_subst[term1] = term2
            return new_subst

    # Step 1c: If term2 is a variable
    elif isinstance(term2, Variable):
        if occurs_check(term2, term1, subst):
            return None  # FAILURE
        else:
            new_subst = subst.copy()
            new_subst[term2] = term1
            return new_subst

    # Step 1d: Both are constants but not equal
    elif isinstance(term1, Constant) or isinstance(term2, Constant):
        return None  # FAILURE

    # Step 2: Check if both are predicates with same name
    elif isinstance(term1, Predicate) and isinstance(term2, Predicate):
```

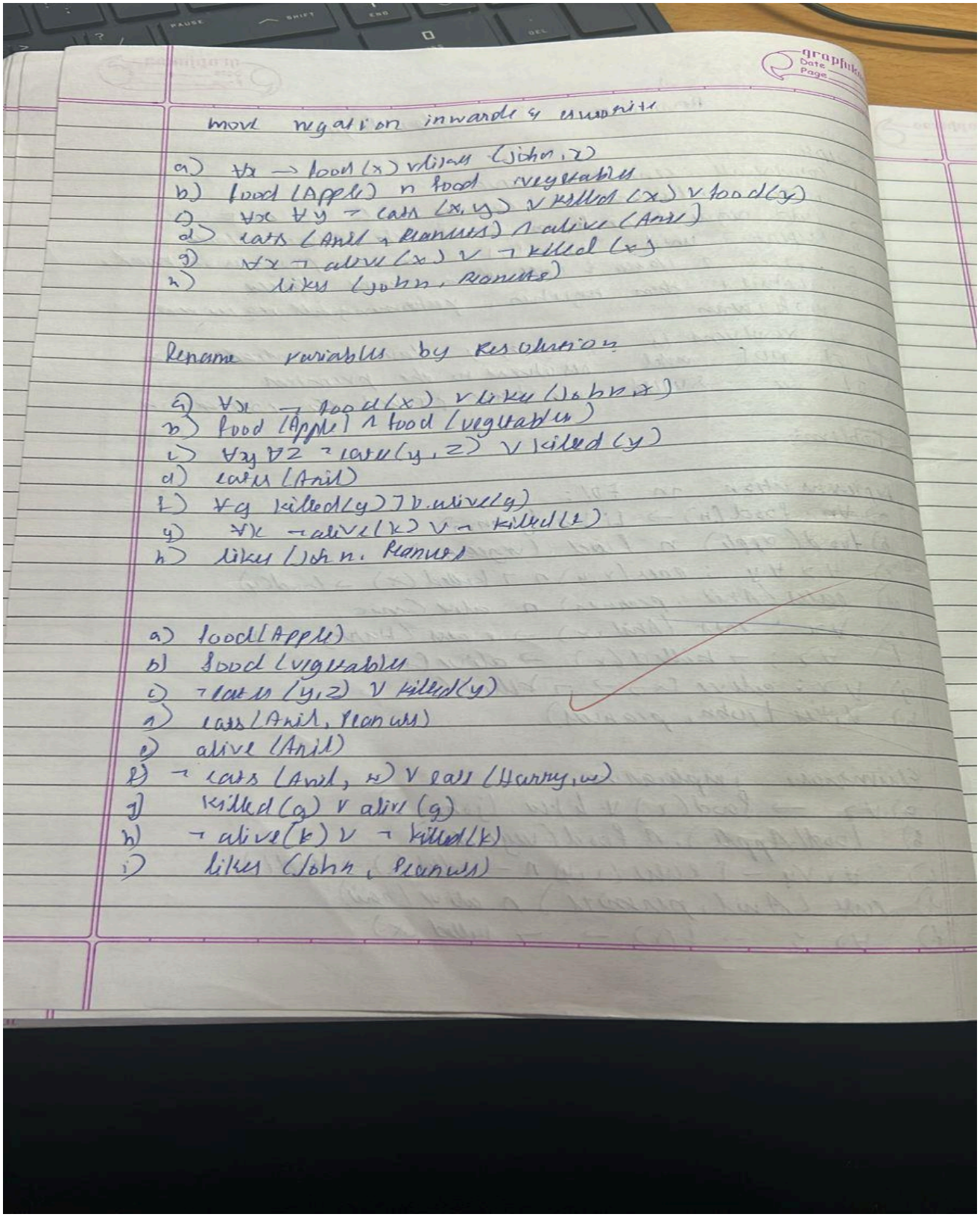
```
        if term1.name != term2.name:
            return None  # FAILURE

        # Step 3: Check if they have same number of arguments
        if len(term1.args) != len(term2.args):
            return None  # FAILURE

        # Step 4 & 5: Unify arguments recursively
        current_subst = subst.copy()
        for arg1, arg2 in zip(term1.args, term2.args):
            current_subst = unify(arg1, arg2, current_subst)
            if current_subst is None:  # If unification fails
                return None

        return current_subst

    else:
        return None  # FAILURE

def print_substitution(subst):
    """Pretty print substitution set"""
    if subst is None:
        print("FAILURE: Unification failed")
    elif not subst:
        print("NIL: Terms are already unified")
    else:
        print("Substitution:")
        for var, term in subst.items():
            print(f"  {var} -> {term}")

def parse_term(term_str):
    """Parse a string representation of a term into Term objects"""
    term_str = term_str.strip()

    # Check if it's a predicate (contains parentheses)
    if '(' in term_str:
        paren_idx = term_str.index('(')
        pred_name = term_str[:paren_idx].strip()

        # Extract arguments between parentheses
        args_str = term_str[paren_idx+1:term_str.rindex(')')].strip()

        # Split arguments by comma (handle nested predicates)
        args = []
        depth = 0
        current_arg = ""
        for char in args_str:
            if char == ',' and depth == 0:
                args.append(parse_term(current_arg))
                current_arg = ""
            else:
                if char == '(':
                    depth += 1
                elif char == ')':
                    depth -= 1
                current_arg += char

        if current_arg.strip():
```

```
            args.append(parse_term(current_arg))

        return Predicate(pred_name, args)

    # Check if it's a variable (lowercase first letter or starts with ?)
    elif term_str[0].islower() or term_str[0] == '?':
        return Variable(term_str)

    # Otherwise it's a constant (uppercase first letter)
    else:
        return Constant(term_str)

def run_interactive():
    """Interactive mode for user input"""
    print("=== Unification Algorithm (Interactive Mode) ===")
    print("Enter terms to unify. Use:")
    print("  - Variables: lowercase letters (x, y, z) or ?x, ?y")
    print("  - Constants: uppercase letters (John, Mary, A)")
    print("  - Predicates: Name(arg1, arg2, ...) e.g., P(x, y)")
    print("  - Type 'quit' to exit\n")

    while True:
        print("-" * 50)
        term1_str = input("Enter first term: ").strip()

        if term1_str.lower() == 'quit':
            print("Exiting...")
            break

        term2_str = input("Enter second term: ").strip()

        if term2_str.lower() == 'quit':
            print("Exiting...")
            break

        try:
            term1 = parse_term(term1_str)
            term2 = parse_term(term2_str)

            print(f"\nUnifying: {term1} and {term2}")
            result = unify(term1, term2)
            print_substitution(result)
            print()

        except Exception as e:
            print(f"Error parsing terms: {e}")
            print("Please check your input format.\n")

def run_examples():
    """Run predefined examples"""
    print("=== Unification Algorithm Examples ===\n")

    # Example 1: Unifying variables
    print("Example 1: Unify(x, y)")
    x = Variable('x')
    y = Variable('y')
    result = unify(x, y)
    print_substitution(result)
```

```
    print()

    # Example 2: Unifying variable with constant
    print("Example 2: Unify(x, John)")
    x = Variable('x')
    john = Constant('John')
    result = unify(x, john)
    print_substitution(result)
    print()

    # Example 3: Unifying predicates
    print("Example 3: Unify(P(x, y), P(John, z))")
    p1 = Predicate('P', [Variable('x'), Variable('y')])
    p2 = Predicate('P', [Constant('John'), Variable('z')])
    result = unify(p1, p2)
    print_substitution(result)
    print()

    # Example 4: Unifying complex predicates
    print("Example 4: Unify(P(x, f(y)), P(a, f(b)))")
    p1 = Predicate('P', [Variable('x'), Predicate('f', [Variable('y')])])
    p2 = Predicate('P', [Constant('a'), Predicate('f', [Constant('b')])])
    result = unify(p1, p2)
    print_substitution(result)
    print()

    # Example 5: Failure case - occurs check
    print("Example 5: Unify(x, f(x)) - Occurs Check")
    x = Variable('x')
    fx = Predicate('f', [x])
    result = unify(x, fx)
    print_substitution(result)
    print()

    # Example 6: Failure case - different predicates
    print("Example 6: Unify(P(x), Q(x)) - Different Predicates")
    p1 = Predicate('P', [Variable('x')])
    p2 = Predicate('Q', [Variable('x')])
    result = unify(p1, p2)
    print_substitution(result)
    print()

    # Example 7: Failure case - different constants
    print("Example 7: Unify(John, Mary) - Different Constants")
    john = Constant('John')
    mary = Constant('Mary')
    result = unify(john, mary)
    print_substitution(result)

# Main program
if __name__ == "__main__":
    print("Choose mode:")
    print("1. Run predefined examples")
    print("2. Interactive mode (enter your own terms)")

    choice = input("\nEnter choice (1 or 2): ").strip()
    print()
```

```
    if choice == '1':
        run_examples()
    elif choice == '2':
        run_interactive()
    else:
        print("Invalid choice. Running examples by default...\n")
        run_examples()
```

program 9: Create a knowledge base consisting of first order logic statements and prove the given query using Resolution



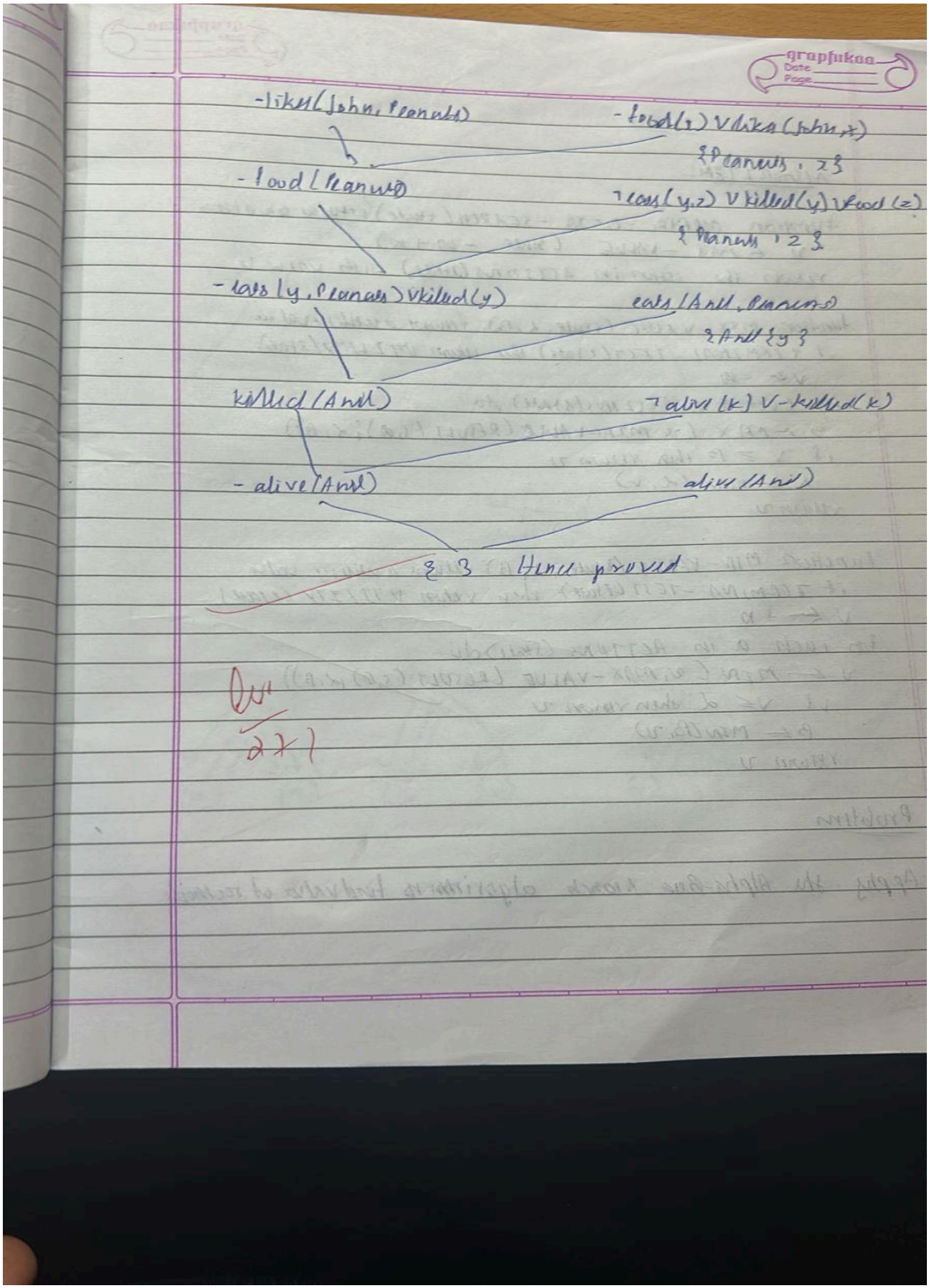
Move negation inwards & rename variables

a) $\forall x \rightarrow food(x) \vee likes(John, x)$
b) food(Apple) ∧ food(vegetables)
c) $\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$
d) eats(Anil, Peanuts) ∧ alive(Anil)
e) $\forall x \neg alive(x) \vee \neg killed(x)$
f) likes(John, Peanuts)

Rename variables by Resolution

a) $\forall x \neg food(x) \vee likes(John, x)$
b) food(Apple) ∧ food(vegetables)
c) $\forall y \forall z \neg eats(y, z) \vee killed(y)$
d) eats(Anil)
e) $\forall g$ killed(g) ⇒ alive(g)
g) $\forall k \neg alive(k) \vee \neg killed(k)$
h) likes(John, Peanuts)

a) food(Apple)
b) food(vegetables)
c) $\neg eats(y, z) \vee killed(y)$
d) eats(Anil, Peanuts)
e) alive(Anil)
f) $\neg eats(Anil, x) \vee eats(Harry, x)$
g) killed(g) ∨ alive(g)
h) $\neg alive(k) \vee \neg killed(k)$
i) likes(John, Peanuts)


-likes(John, Peanuts)
-food(Peanuts)
-eats(y,Peanuts) V killed(y)
killed(Anil)
-alive(Anil)
{} Hence proved

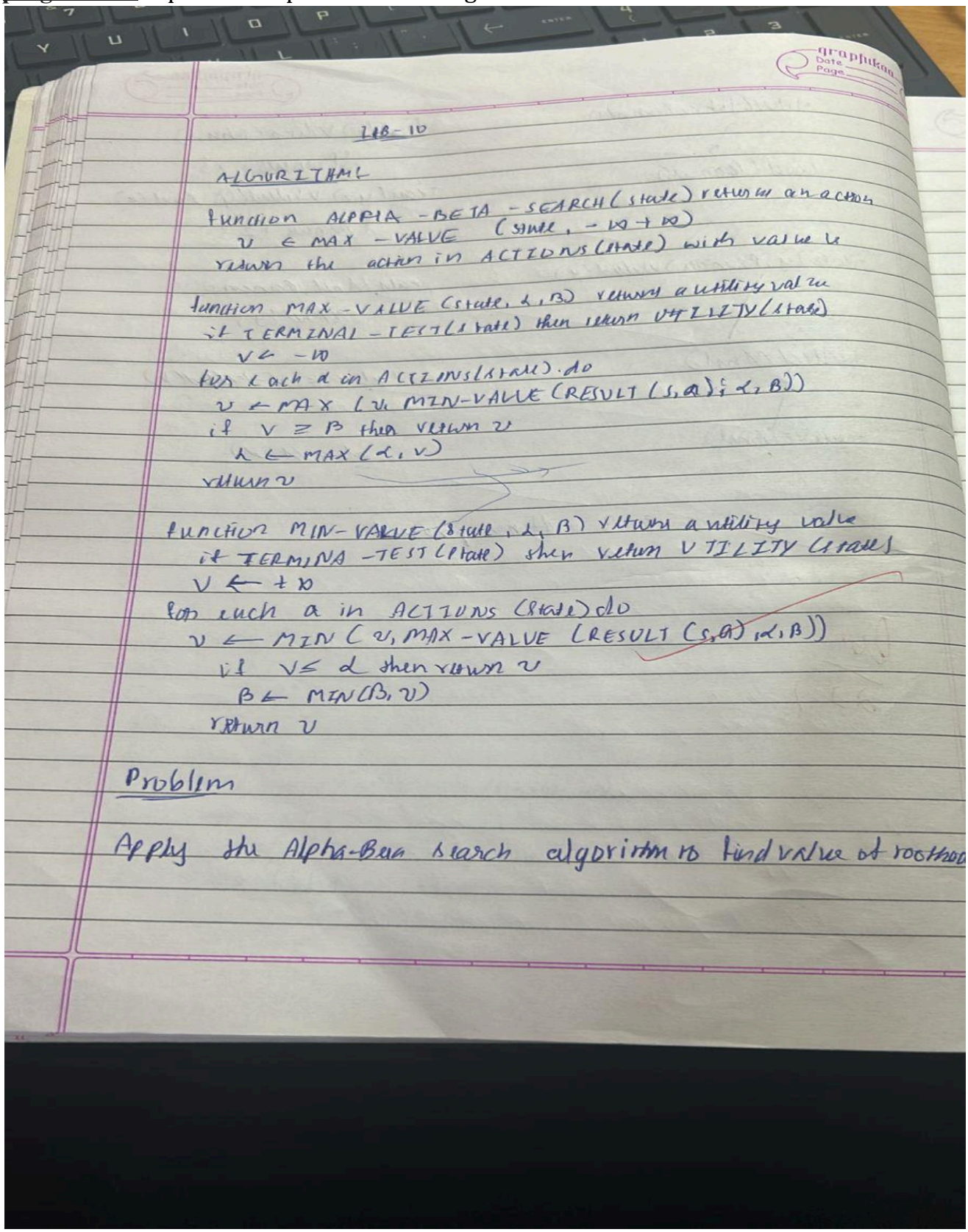
code

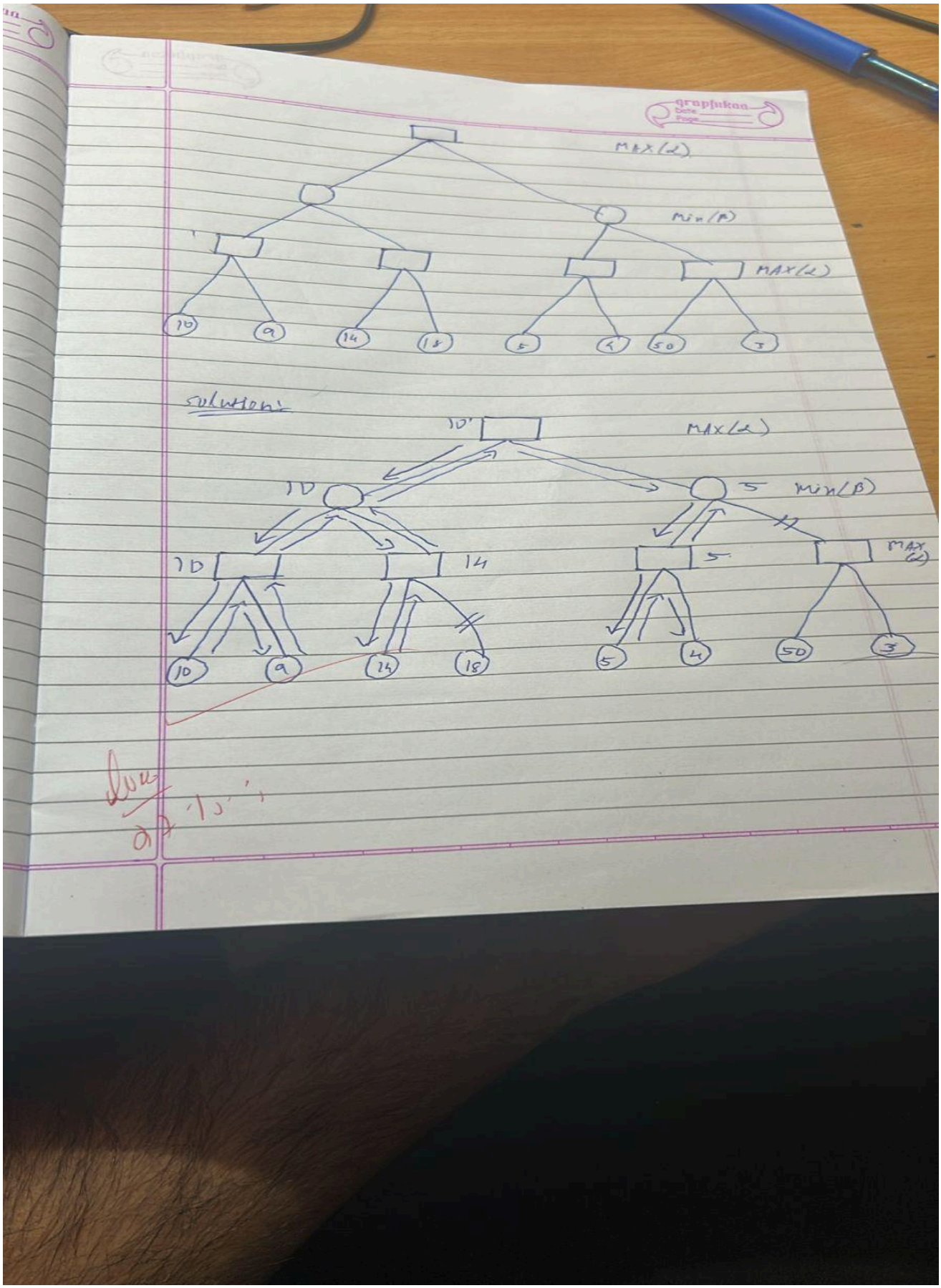
```
from copy import deepcopy

# ------------------------------
# Utility functions
# ------------------------------
def substitute(clause, subs):
    """Apply substitutions to a clause."""
    new_clause = []
    for literal in clause:
        pred, args, neg = literal
        new_args = []
        for a in args:
            if a in subs:
                new_args.append(subs[a])
            else:
                new_args.append(a)
        new_clause.append((pred, new_args, neg))
    return new_clause

def unify(x, y, subs=None):
    """Unify two literals/terms."""
    if subs is None:
        subs = {}
    if x == y:
        return subs
    if isinstance(x, str) and x[0].islower():
        return unify_var(x, y, subs)
    elif isinstance(y, str) and y[0].islower():
        return unify_var(y, x, subs)
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None
        for a, b in zip(x[1], y[1]):
            subs = unify(a, b, subs)
            if subs is None:
                return None
        return subs
    else:
        return None

def unify_var(var, x, subs):
    if var in subs:
        return unify(subs[var], x, subs)
    elif x in subs:
        return unify(var, subs[x], subs)
    elif occur_check(var, x, subs):
        return None
    else:
        subs[var] = x
        return subs

def occur_check(var, x, subs):
    if var == x:
        return True
    elif isinstance(x, str) and x in subs:
        return occur_check(var, subs[x], subs)
```

```
    elif isinstance(x, tuple):
        return any(occur_check(var, arg, subs) for arg in x[1])
    return False


# ------------------------------
# Resolution core
# ------------------------------
def resolve(clause1, clause2):
    """Try all pairs of literals to resolve."""
    resolvents = []
    for lit1 in clause1:
        for lit2 in clause2:
            if lit1[0] == lit2[0] and lit1[2] != lit2[2]:  # same
predicate, opposite polarity
                subs = unify(tuple(lit1[:2]), tuple(lit2[:2]))
                if subs is not None:
                    new_c1 = substitute(clause1, subs)
                    new_c2 = substitute(clause2, subs)
                    new_clause = [l for l in new_c1 if l != lit1] + [l for
l in new_c2 if l != lit2]
                    # remove duplicates
                    new_clause = [x for i, x in enumerate(new_clause) if x
not in new_clause[:i]]
                    resolvents.append(new_clause)
    return resolvents


def pl_resolution(kb, query):
    clauses = deepcopy(kb)
    # add negated query
    neg_query = [(query[0], query[1], not query[2])]
    clauses.append(neg_query)

    print("Initial clauses:")
    for i, c in enumerate(clauses):
        print(f"C{i+1}: {pretty_clause(c)}")
    print("\nResolution steps:\n")

    new = []
    while True:
        n = len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in
range(i + 1, n)]
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            for r in resolvents:
                if not r:
                    print("Derived empty clause {} ->
CONTRADICTION.".format(pretty_clause(r)))
                    return True
                if r not in clauses and r not in new:
                    new.append(r)
                    print(f"Resolved {pretty_clause(ci)} and
{pretty_clause(cj)} -> {pretty_clause(r)}")
        if all(r in clauses for r in new):
            return False
        clauses.extend(new)
```

```
# ------------------------------
# Pretty printer
# ------------------------------
def pretty_clause(clause):
    if not clause:
        return "{}"
    out = []
    for pred, args, neg in clause:
        s = f"{pred}({', '.join(args)})"
        if neg:
            s = "¬" + s
        out.append(s)
    return " ∨ ".join(out)


# ------------------------------
# Example Knowledge Base
# ------------------------------
if __name__ == "__main__":
    # KB Clauses:
    # 1. ¬parent(x,y) ∨ ancestor(x,y)
    # 2. ¬ancestor(x,y) ∨ ¬ancestor(y,z) ∨ ancestor(x,z)
    # 3. parent(tom,bob)
    # 4. parent(bob,alice)
    KB = [
        [('parent', ['x', 'y'], True), ('ancestor', ['x', 'y'], False)],
        [('ancestor', ['x', 'y'], True), ('ancestor', ['y', 'z'], True),
('ancestor', ['x', 'z'], False)],
        [('parent', ['tom', 'bob'], False)],
        [('parent', ['bob', 'alice'], False)]
    ]

    # Query: ancestor(tom, alice)
    QUERY = ('ancestor', ['tom', 'alice'], False)

    result = pl_resolution(KB, QUERY)
    print("\nResult:", "PROVED ✅" if result else "NOT PROVABLE ❌")
```

program 10: Implement Alpha-Beta Pruning.




MAX(α)
Min(β)
MAX(α)
10
9
14
18
5
4
50
3
solution:
10
MAX(α)
10
5
Min(β)
10
14
5
MAX(α)
10
9
14
18
5
4
50
3

code:

```
import math

def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta,
max_depth):
    if depth == max_depth:
        return values[node_index]

    if maximizing_player:
        best = -math.inf
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, False, values,
alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                print(f"Pruned at depth {depth}, node {node_index},
α={alpha}, β={beta}")
                break
        return best
    else:
        best = math.inf
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, True, values,
alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                print(f"Pruned at depth {depth}, node {node_index},
α={alpha}, β={beta}")
                break
        return best


values = [10, 9, 14, 18, 5, 4, 50, 3]
max_depth = 3

print("ALPHA-BETA PRUNING PROCESS\n")
optimal_value = alpha_beta(0, 0, True, values, -math.inf, math.inf,
max_depth)
print("\nOptimal value (Root Node):", optimal_value)
```