VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sam Manu Jacob (1BM23CS291)

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 **Sam Manu Jacob (1BM23CS291),** 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. Seema Patil Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

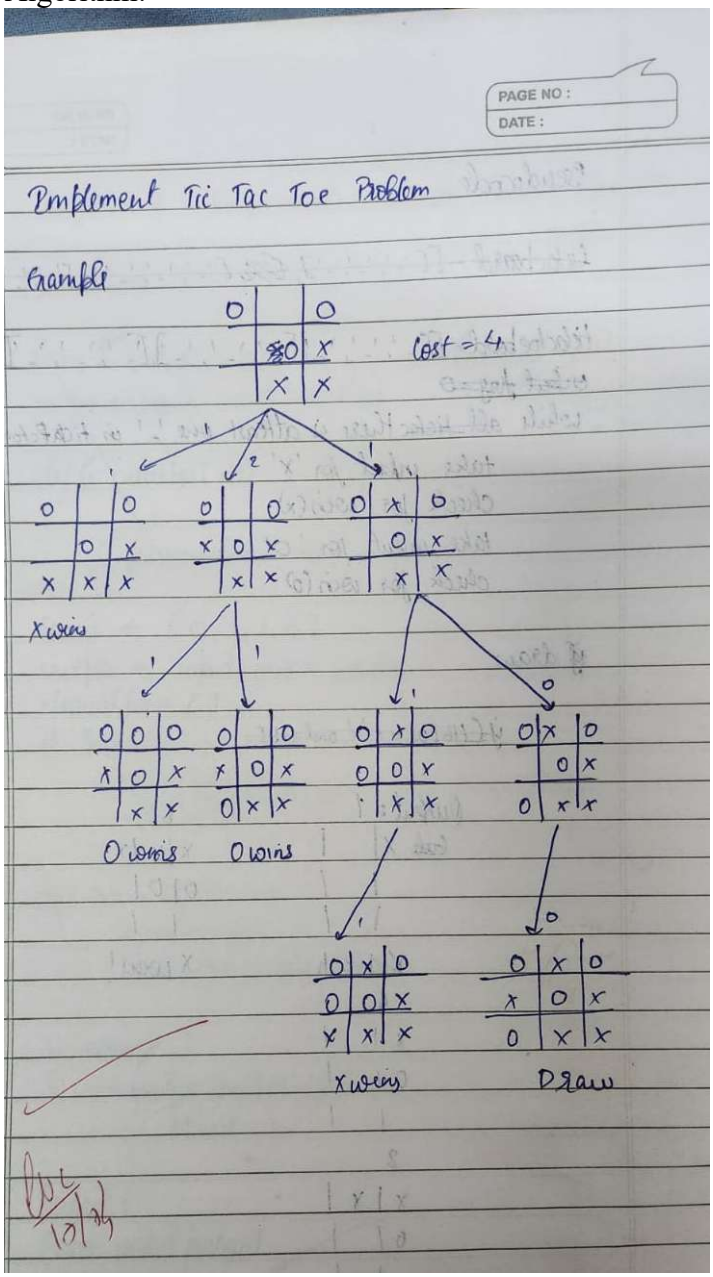
Index

Github Link: https://github.com/SamManuJacob/1BM23CS291_AILAB

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:



PAGE NO :
DATE :

Pseudocode

~~tictacboard = [['-','-','-'], ['-','-','-'], ['-','-','-']]~~

tictactoeboard = [['-','-','-'],['-','-','-'],['-','-','-']]
~~input flag = 0~~
while ~~all tictac~~ there is atleast one '-' in tictactoeboard:
    take input for 'X'
    check for win(x)
    take input for 'O'
    check for win(O)

draw

~~if (ttb[0] == 'x' and ttb[~~

Output: 1

X | |
 | |
 | |

4

X | |
O | |
 | |

2

X | X |
O | |
 | |

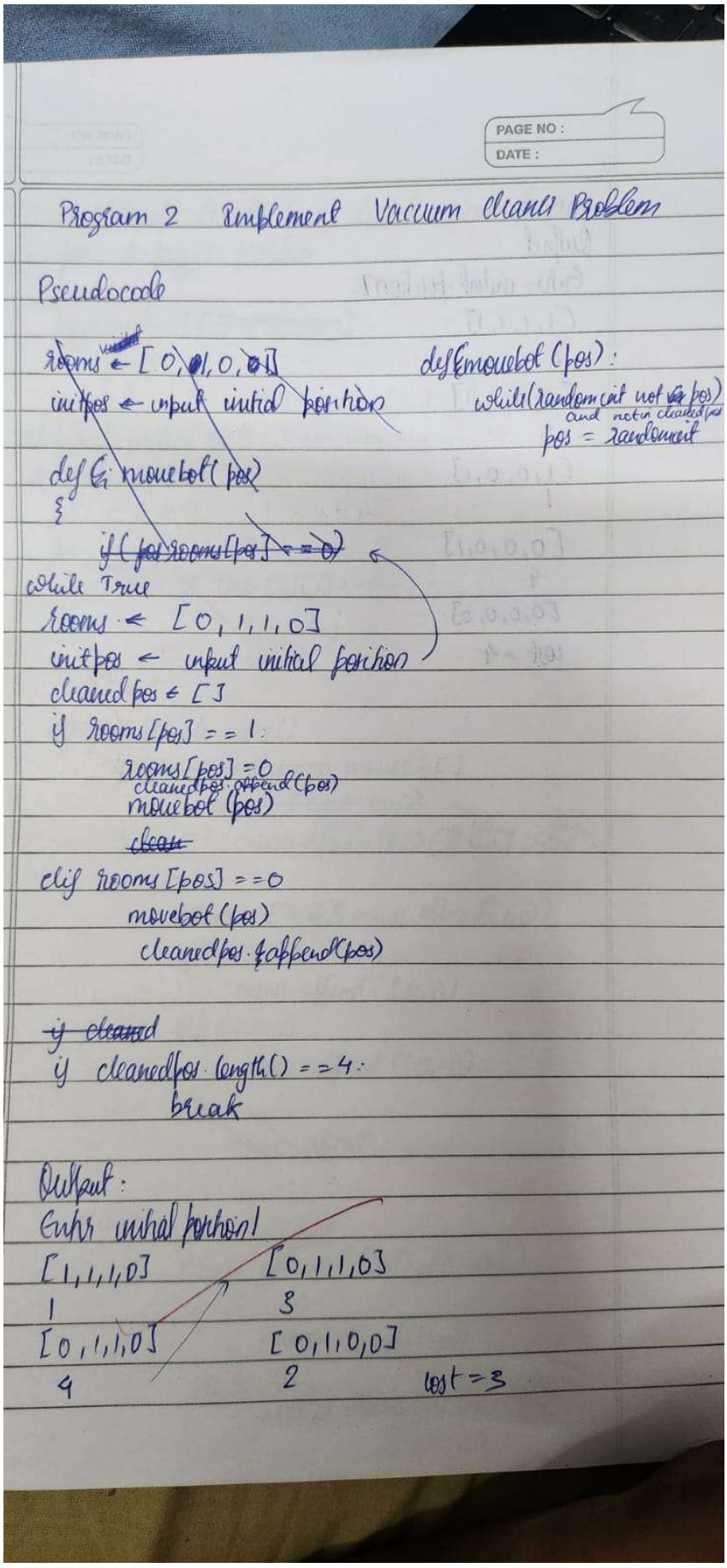
5

X | X |
O | O |
 | |

6

X | X | X
O | O |
 | |

X wins!



PAGE NO :
DATE :

Program 2 Implement Vacuum cleaner Problem

Pseudocode

```
rooms ← [0,1,0,0]
initpos ← input initial position

def E movebot(pos)
{
    if(rooms[pos] == 0)
while True
rooms ← [0,1,1,0]
initpos ← input initial position
cleanedpos ← []
if rooms[pos] == 1:
    rooms[pos] = 0
    cleanedpos.append(pos)
    movebot(pos)
    clean
elif rooms[pos] == 0
    movebot(pos)
    cleanedpos.append(pos)

if cleaned
if cleanedpos.length() == 4:
    break
```

```
def movebot(pos):
    while(randomint not in pos)
          and not in cleanedpos
        pos = randomint
```

Output:

Enter initial position

[1,1,1,0]
1

[0,1,1,0]
4

[0,1,1,0]
3

[0,1,0,0]
2

cost = 3

Code:

```
def print_board(board):
    print("\n")
    for i in range(3):
        print(" | ".join(board[i*3:(i+1)*3]))
        if i < 2:
            print("-" * 10)
    print("\n")

def check_winner(board, player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]
    for combo in win_conditions:
        count=0
        for pos in combo:
            if board[pos]==player:
                count+=1
        if count==3:
            return True
    return False

board = [" "] * 9
current_player = "X"
print_board(board)

while True:
    while True:
        pos = int(input(f"Player {current_player}, enter your move (1-9): ")) - 1
        if 0 <= pos <= 8 and board[pos] == " ":
            board[pos] = current_player
            break
        else:
            print("Invalid move. Try again.")

    print_board(board)

    if check_winner(board, current_player):
        print(f"Player {current_player} wins!")
        break
    if " " not in board:
        print("It's a draw!")
        break

    # Switch players
```

```
    current_player = "O" if current_player == "X" else "X"
```

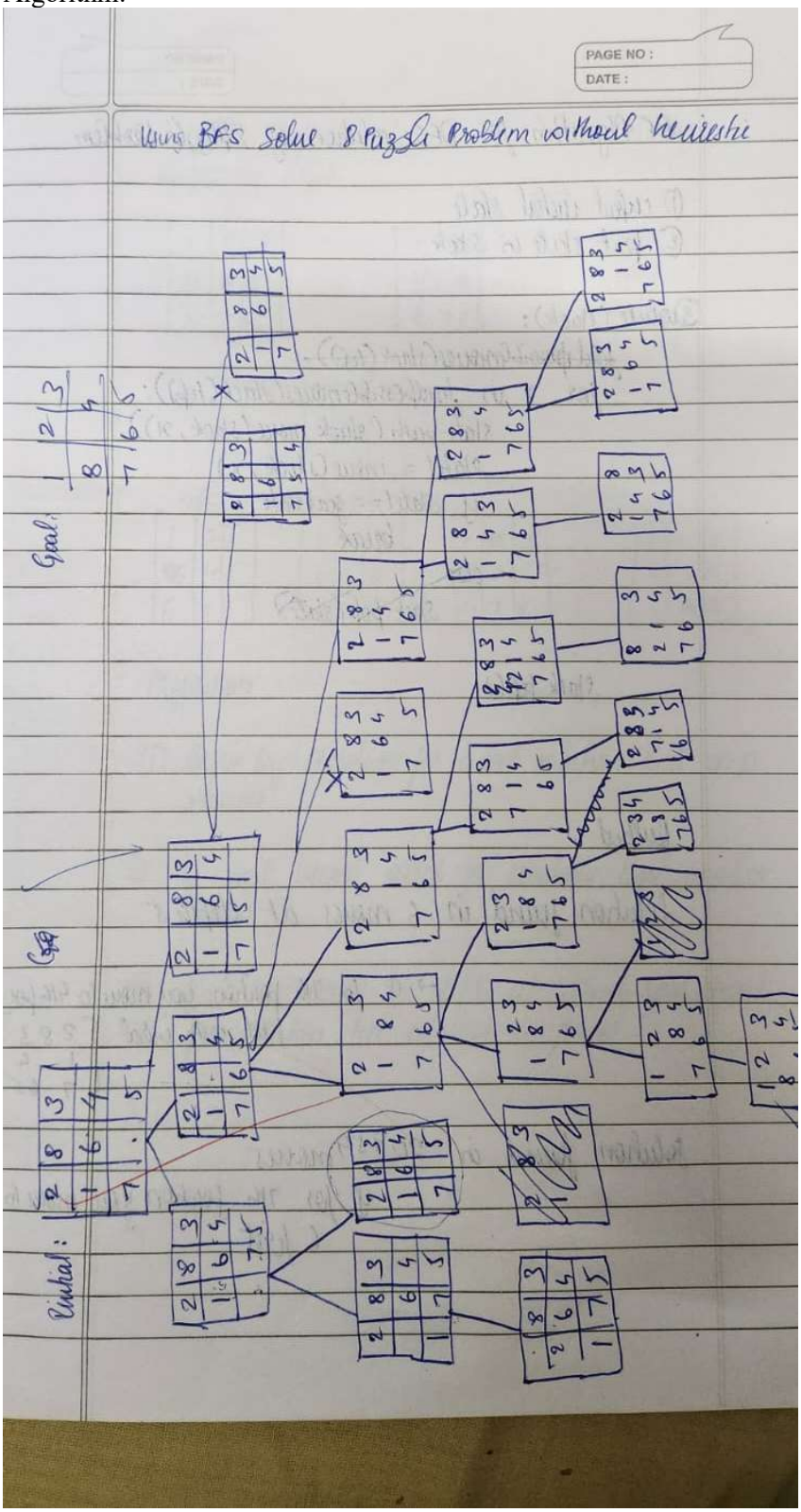
Vacuum Cleaner:

```
import random
rooms=[1,1,1,1]
botpos =(int(input("Enter Initial Position"))-1)
cleanedpos=[]
cost=0

def movebot(pos):

    while True:
        n= random.randint(0,3)
        if n != pos and n not in cleanedpos:
            pos = n
            break
    return pos

while True:
    print(str(rooms))
    print(botpos+1)

    if rooms[botpos]==1:

        rooms[botpos]=0
        cleanedpos.append(botpos)
        cost+=1
        if len(cleanedpos) == 4:
            break
        botpos=movebot(botpos)

    elif rooms[botpos]==0:
        cleanedpos.append(botpos)
        if len(cleanedpos) == 4:
            break
        botpos = movebot(botpos)




print("cost="+str(cost))
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:



PAGE NO :
DATE :

Algorithm for DFS solution of 8 puzzle problem

① input initial state
② put state in stack

③ solve (stack):
    ~~find possible moves (stack (top)):~~
    for x in findpossiblemoves (stack (top)):
        stack.push (~~stack~~ move (stack, x))
        state1 = move (stack, x)
        if state1 == goalstate:
            break
        else:
            stack.pop (state)

    stack.pop()

Output

solution found in 6 moves at depth 5

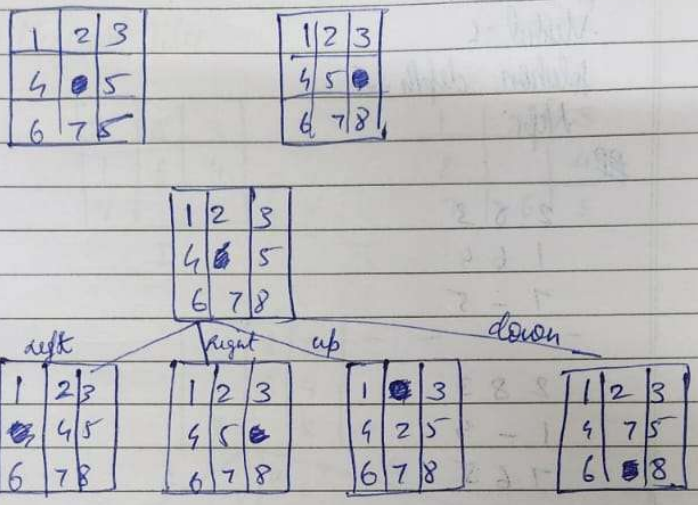
→ if for 7th position you move to 4th for first with initial $\begin{bmatrix} 2 & 8 & 3 \\ 1 & - & 4 \\ 7 & 6 & 5 \end{bmatrix}$

solution found in 20959 moves

if for 7th position you move to 6 first

PAGE NO :
DATE :

Iterative Deepening Search (IDS) or Iterative Deepening depth



Algorithm

① Begin by searching for a ~~not~~ solution that is 0 moves

② If not found search for solution that involve 1 move

③ Keep increasing the depth at which you search for solution till solution is found.

Code:

```
print("solution by Sam Manu Jacob- 1BM23CS291")
def find_possible_moves(state):
    index = state.index('_')
    moves = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [6, 8, 4],
        8: [5, 7],
    }
    return moves.get(index, [])

def dfs(initial_state, goal_state, max_depth=50):
    stack = [(initial_state, [], 0)]
    visited = {tuple(initial_state)}
    states_explored = 0
    printed_depths = set()

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

        if depth > max_depth:
            continue

        if depth not in printed_depths:
            print(f"\n--- Depth {depth} ---")
            printed_depths.add(depth)

        states_explored += 1
        print(f"State #{states_explored}: {current_state}")

        if current_state == goal_state:
            print(f"\n Goal reached at depth {depth} after exploring {states_explored} states.\n")
            return path

        possible_moves_indices = find_possible_moves(current_state)

        for move_index in reversed(possible_moves_indices):  # Reverse for DFS order
            next_state = list(current_state)
            blank_index = next_state.index('_')
            next_state[blank_index], next_state[move_index] = next_state[move_index],
next_state[blank_index]
```

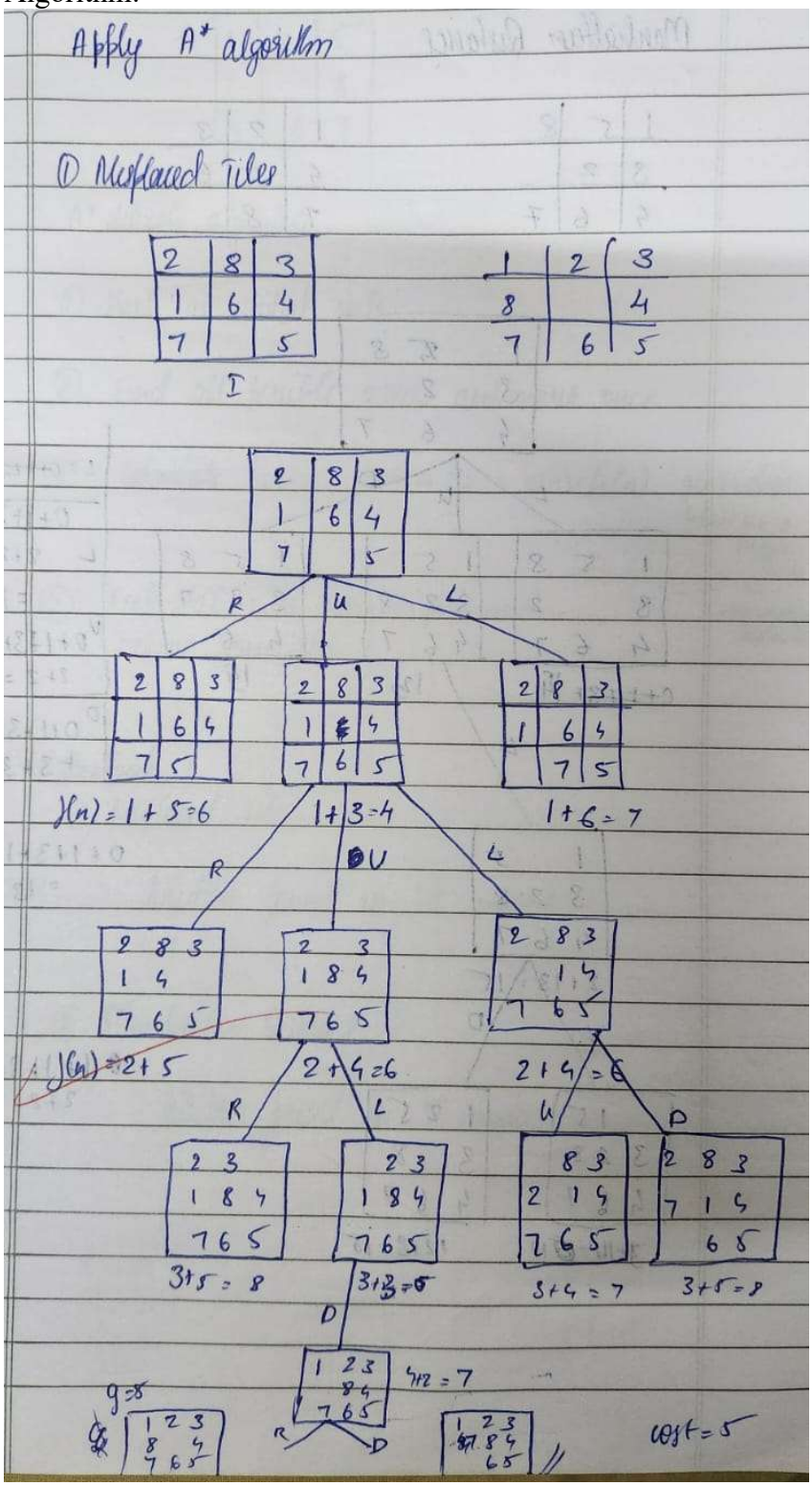
```
            if tuple(next_state) not in visited:
                visited.add(tuple(next_state))
                stack.append((next_state, path + [next_state], depth + 1))

    print(f"\n Goal state not reachable within depth {max_depth}. Explored {states_explored}
states.\n")
    return None

initial_state = [2, 8, 3, 1, 6, 4, 7, '_', 5]
goal_state    = [1, 2, 3, 8, '_', 4, 7, 6, 5]


solution_path = dfs(initial_state, goal_state, max_depth=50)


if solution_path is None:
    print("No solution found.")
else:
    print("Solution path:")
    for step, state in enumerate(solution_path, start=1):
        print(f"Step {step}: {state}")


Iterative Deepening:
import copy
print("Solution by Sam")
goal = [[1, 2, 3],
        [8, 0, 4],
        [7, 6, 5]]


moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

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

def is_goal(state):
    return state == goal

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    for dx, dy in moves:
```

```
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = copy.deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

def print_state(state):
    for row in state:
        print(" ".join(str(num) if num != 0 else "_" for num in row))
    print("----------")

def dfs_limited(state, path, depth, limit, visited, current_depth):
    if depth == limit:

        return None



    if is_goal(state):
        return path + [state]

    visited.add(state_to_tuple(state))

    for neighbor in get_neighbors(state):
        key = state_to_tuple(neighbor)
        if key not in visited:
            result = dfs_limited(neighbor, path + [state], depth + 1, limit, visited, current_depth)
            if result:
                return result
    return None

def iddfs(start_state, max_depth=50):
    for limit in range(max_depth + 1):

        visited = set()
        path = dfs_limited(start_state, [], 0, limit, visited, 0)
        if path:
            print("Goal reached!")
            print("Visited:", len(visited))
            print("Solution depth:", len(path) - 1)
            print("Steps:")
            for step in path:
                print_state(step)
```

```
            return
    print("No solution found within depth limit.")

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

iddfs(start_state, max_depth=20)
```

Program 3
Implement A* search algorithm

Algorithm:

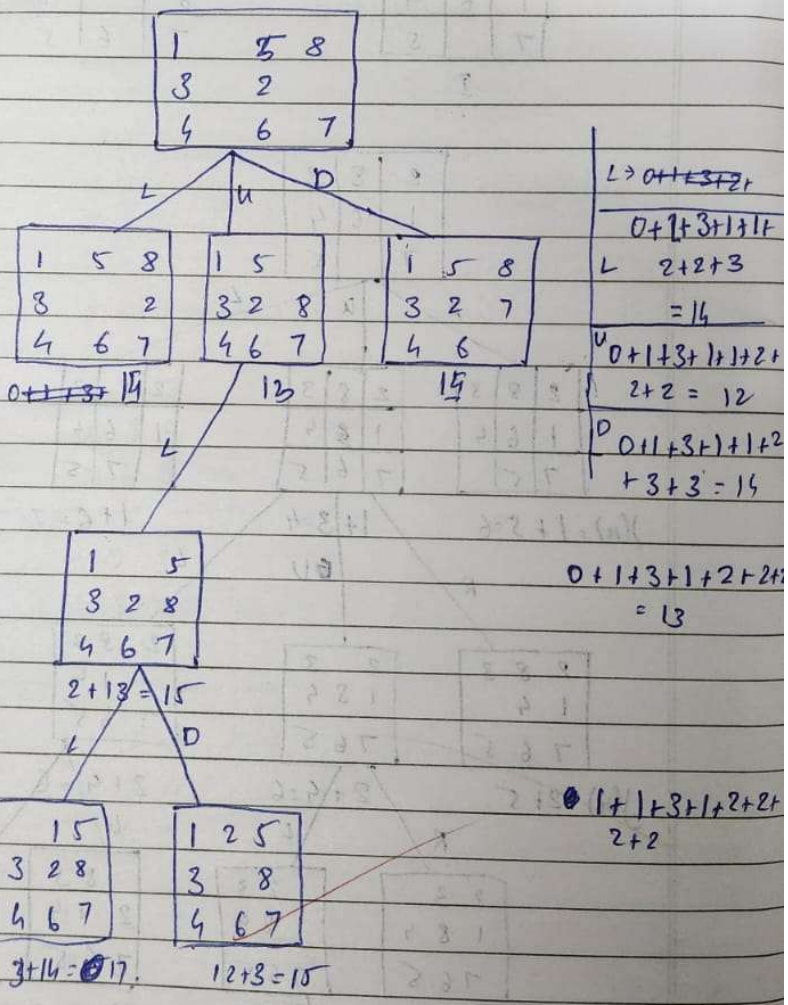


PAGE NO :
DATE :

Manhattan Distance

1	5	8
3	2	
4	6	7

1	2	3
4	5	6
7	8	



PAGE NO :
DATE :

1	2	5
3		8
4	6	7

A* search algorithm

① Start at initial state

② Find all possible moves and move once

③ Calculate heuristic cost → $f(n) = g(n) + h(n)$ $g(n)$ = depth, $h(n)$ = no. of misplaced tiles or manhattan distance

④ Pick state with minimum $f(n)$ and repeat from step 4

Output

① Misplaced Tile

Solution found in 5 moves

② Manhattan Distance

Solution found in 5 moves

Code:

```
import heapq
print("Solution by Sam")
GOAL_STATE = (
    (1, 2, 3),
    (8, 0, 4),
    (7, 6, 5)
)

def heuristic(state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1
    return misplaced

def get_neighbors(state):
    neighbors = []
    x = y = -1
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break
        if x != -1:
            break

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx, dy in directions:
        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 reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
```

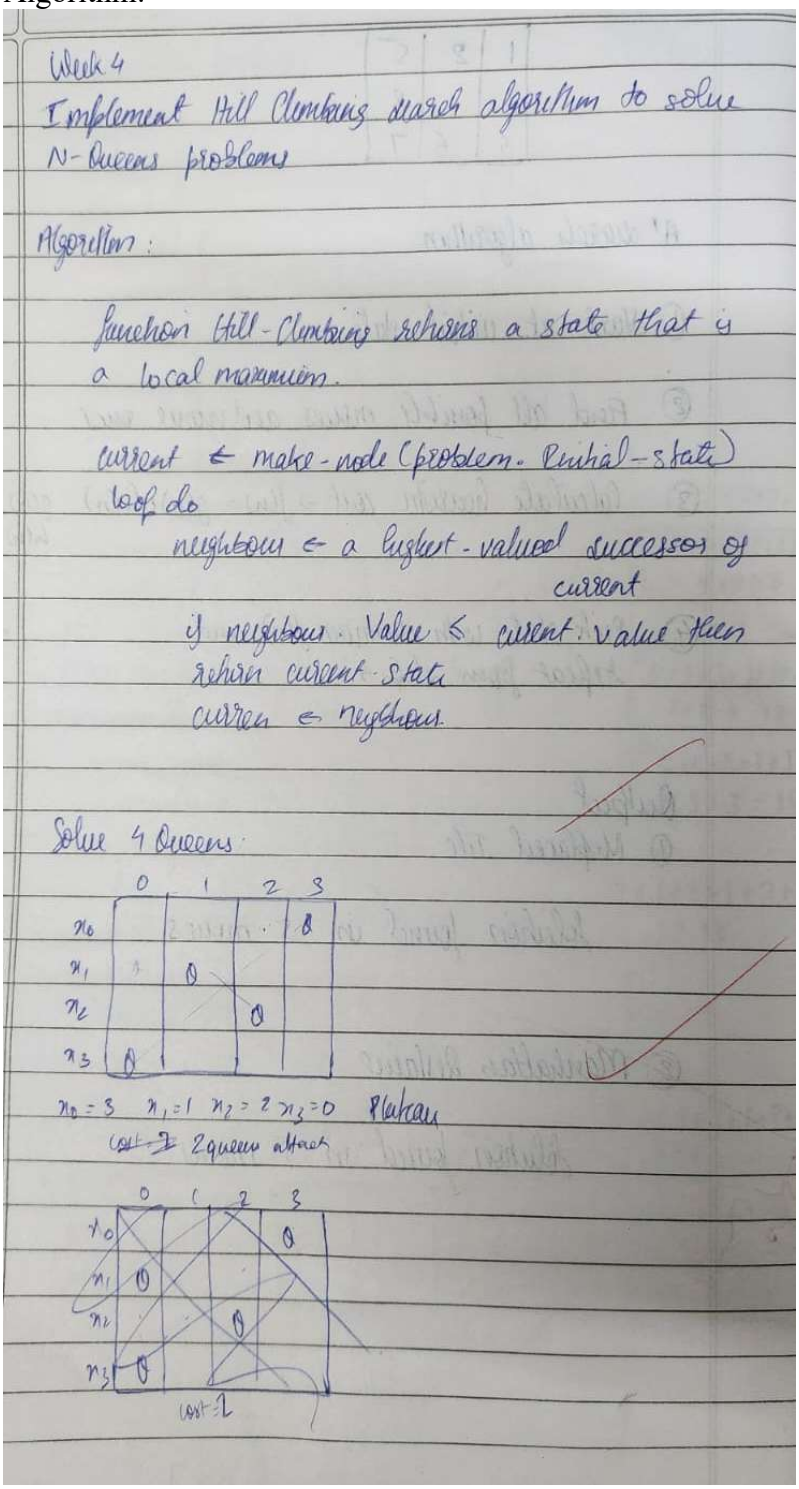
```
    return path

def print_state(state):
    for row in state:
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

def a_star(start_state):
    start_state = tuple(tuple(row) for row in start_state)
    open_set = []
    heapq.heappush(open_set, (heuristic(start_state), 0, start_state))
    came_from = {}
    g_score = {start_state: 0}
    visited = set()

    while open_set:
        f_current, current_g, current = heapq.heappop(open_set)


        print(f"Visited depth (g): {current_g}, f(n): {f_current}")
        print_state(current)

        if current == GOAL_STATE:
            return reconstruct_path(came_from, current)

        visited.add(current)

        for neighbor in get_neighbors(current):
            if neighbor in visited:
                continue

            tentative_g = current_g + 1

            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + heuristic(neighbor)
                heapq.heappush(open_set, (f_score, tentative_g, neighbor))

    return None


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

```
path = a_star(start_state)

if path:
    print(f"Solution found in {len(path) - 1} moves:\n")
    for step in path:
        print_state(step)
else:
    print("No solution found.")
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Week 4

Implement Hill Climbing search algorithm to solve N-Queens problems

Algorithm:

function Hill-Climbing returns a state that is a local maximum.

current ← make-node (problem. Initial-state)

loop do

neighbour ← a highest-valued successor of current

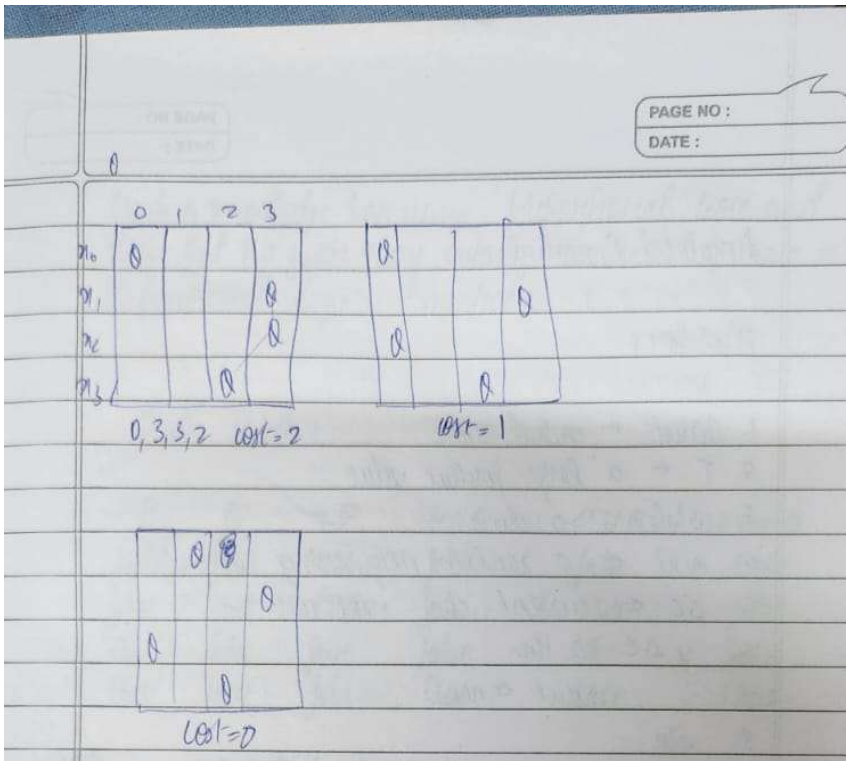
if neighbour. Value ≤ current. value then return current state

current ← neighbour

Solve 4 Queens:

$x_0 = 3$ $x_1 = 1$ $x_2 = 2$ $x_3 = 0$ Plateau

cost = 2 queen attack



Output

Step 0 state = [0,3,3,2], cost = 2
step 1 state = [0,3,0,2], cost = 1
step 2 state = [1,3,0,2], cost = 0

12/2024

Code:

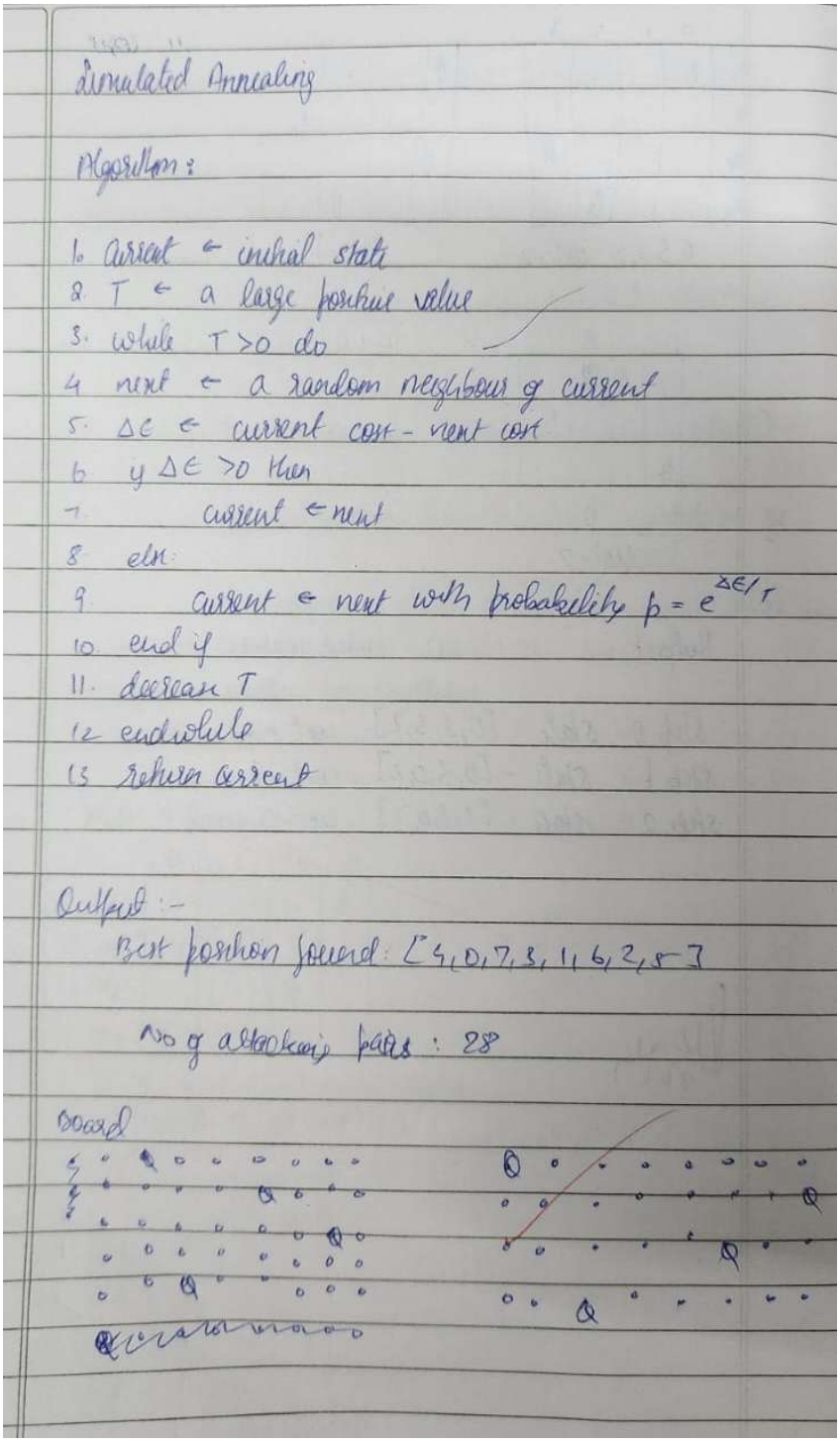
```
import random
import math

def compute_cost(state):
    """Count diagonal conflicts for a permutation-state (one queen per row & column)."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def random_permutation(n):
    arr = list(range(n))
    random.shuffle(arr)
    return arr

def neighbors_by_swaps(state):
    """All neighbors obtained by swapping two columns (keeps permutation property)."""
    n = len(state)
    for i in range(n - 1):
        for j in range(i + 1, n):
            nb = state.copy()
            nb[i], nb[j] = nb[j], nb[i]
            yield nb

def hill_climb_with_restarts(n, max_restarts=None):
    """Hill climbing on permutations with random restart on plateau (no revisits)."""
    visited = set()
    total_states = math.factorial(n)
    restarts = 0

    while True:
        # pick a random unvisited start permutation
        if len(visited) >= total_states:
            raise RuntimeError("All states visited — giving up (no solution found).")

        state = random_permutation(n)
        while tuple(state) in visited:
            state = random_permutation(n)
        visited.add(tuple(state))

        # climb from this start
```

```
        while True:
            cost = compute_cost(state)
            if cost == 0:
                return state, restarts

            # find best neighbor (swap-based neighbors)
            best_neighbor = None
            best_cost = float("inf")
            for nb in neighbors_by_swaps(state):
                c = compute_cost(nb)
                if c < best_cost:
                    best_cost = c
                    best_neighbor = nb

            # if strictly better, move; otherwise it's a plateau/local optimum -> restart
            if best_cost < cost:
                state = best_neighbor
                visited.add(tuple(state))
            else:
                # plateau or local optimum -> restart
                restarts += 1
                if max_restarts is not None and restarts >= max_restarts:
                    raise RuntimeError(f"Stopped after {restarts} restarts (no solution found).")
                break  # go pick a new unvisited start

def format_board(state):
    n = len(state)
    lines = []
    for r in range(n):
        lines.append(" ".join("Q" if state[c] == r else "-" for c in range(n)))
    return "\n".join(lines)

if __name__ == "__main__":
    n = 4
    print("Solution By Sam Manu Jacob - 1BM23CS291")
    solution, restarts = hill_climb_with_restarts(n)
    print("Found solution:", solution)
    print(format_board(solution))
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

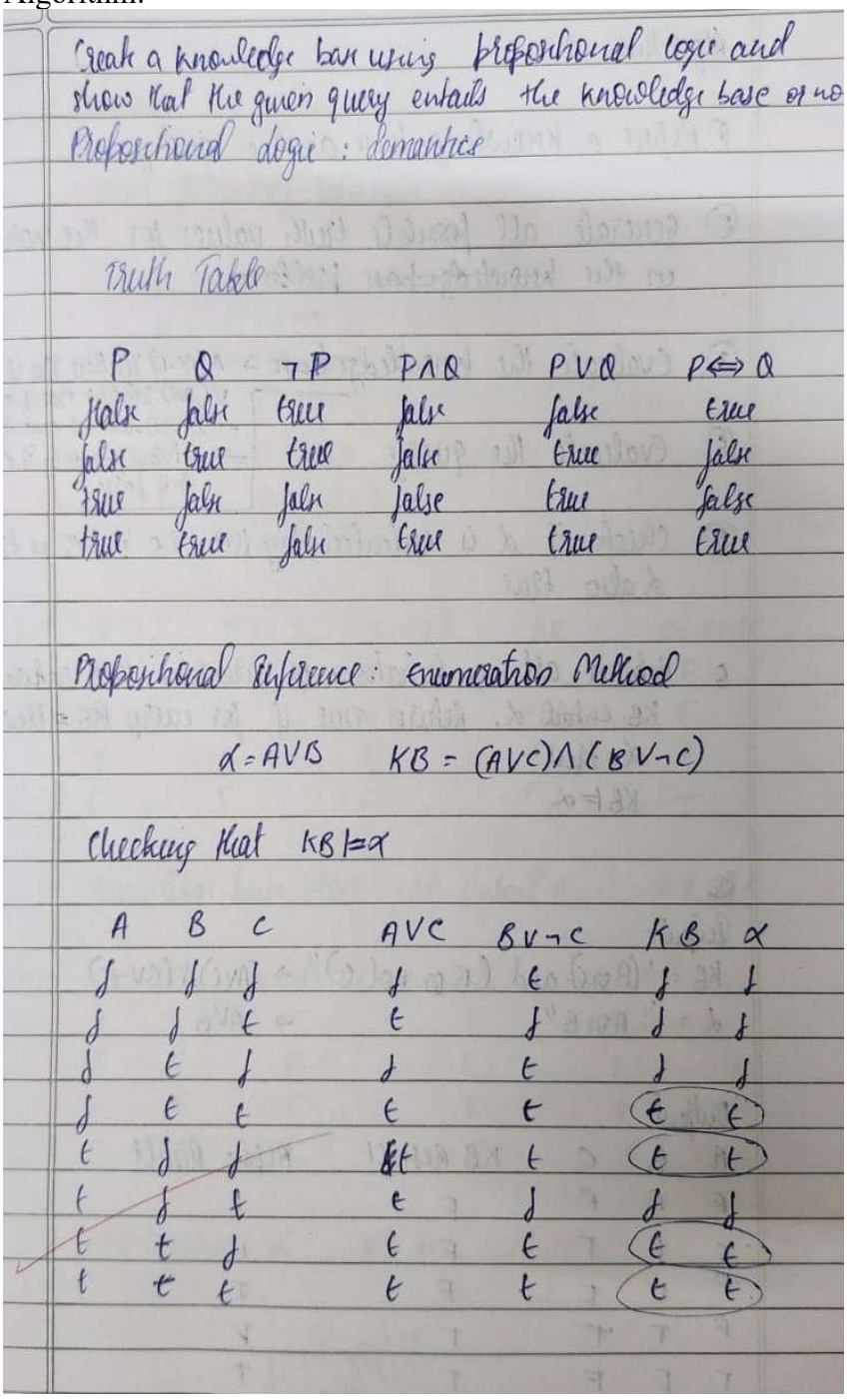
```
import random
import math

def cost(state):

    attacks = 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):
                attacks += 1
    return attacks

def get_neighbor(state):

    neighbor = state[:]
    i, j = random.sample(range(len(state)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

def simulated_annealing(n=8, max_iter=10000, temp=100.0, cooling=0.95):

    current = list(range(n))
    random.shuffle(current)
    current_cost = cost(current)

    temperature = temp
    cooling_rate = cooling

    best = current[:]
    best_cost = current_cost

    for _ in range(max_iter):
        if temperature <= 0 or best_cost == 0:
            break

        neighbor = get_neighbor(current)
        neighbor_cost = cost(neighbor)
        delta = current_cost - neighbor_cost

        if delta > 0 or random.random() < math.exp(delta / temperature):
            current, current_cost = neighbor, neighbor_cost
            if neighbor_cost < best_cost:
                best, best_cost = neighbor[:], neighbor_cost
```

```
        temperature *= cooling_rate

    return best, best_cost

def print_board(state):

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


n = 8
solution, cost_val = simulated_annealing(n, max_iter=20000)
print("Solution by Sam Manu Jacob")
print("Best position found:", solution)
print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
print("\nBoard:")
print_board(solution)
```

Program 6

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

Algorithm:



DATE :

Algorithm:

① Refers a knowledge base and a query

② Generate all possible truth values for the variables in the ~~knowledge base~~ problem

③ Evaluate the knowledge base → ∧ (AND) returns True if both
V (OR) returns true if any one
→ (not) returns true if input
→ returns false only if P is true and Q is false

④ Evaluate the query

⑤ Check if α is entailed by KB, i.e if KB is true α also true

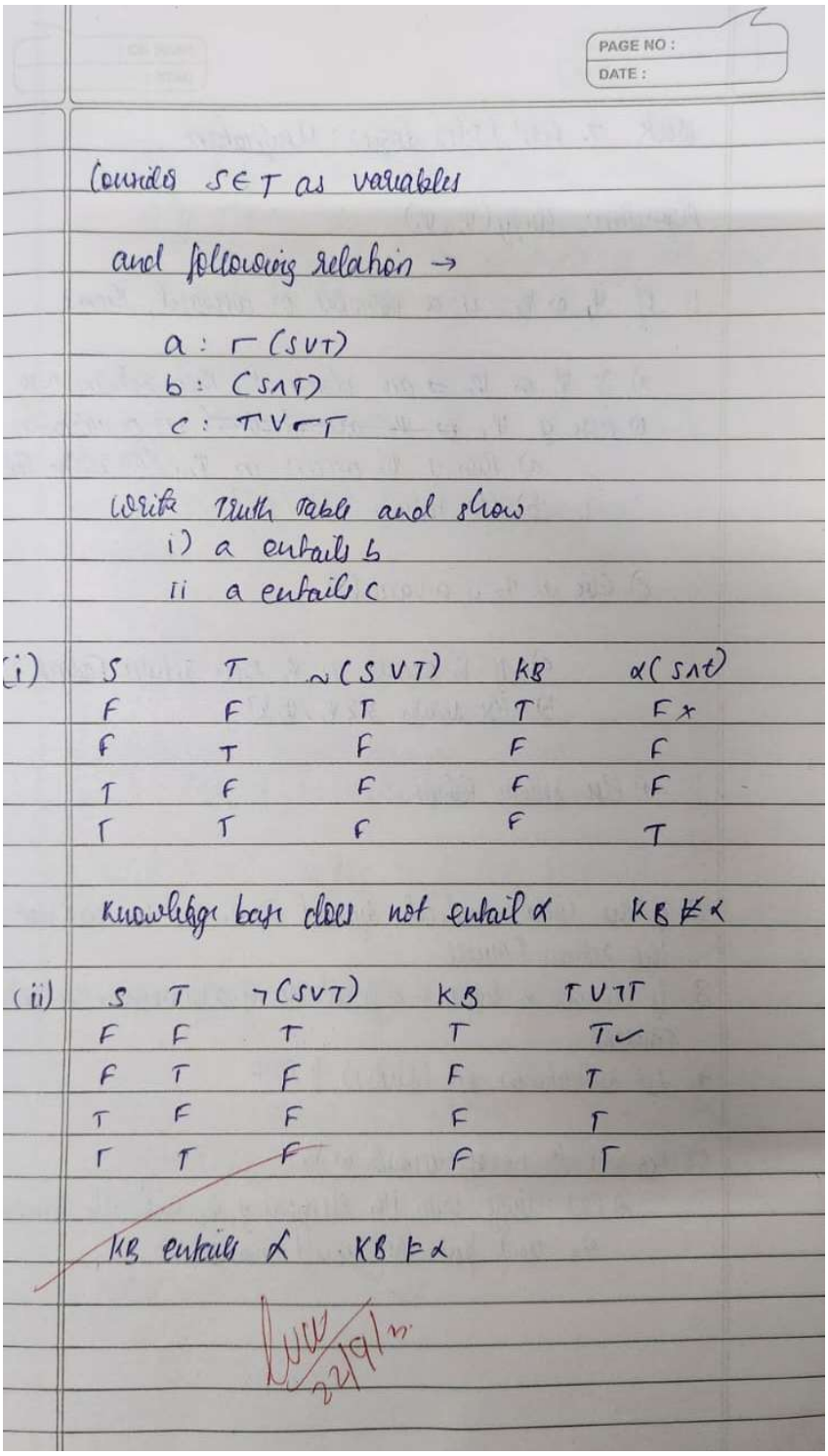
6 Return all combinations of variables that returns KB entails α. Return true if for every KB = True
α = true
KB ⊨ α

Eg

Input:
KB = "(A or C) and (B or not C)" → (A∨C) ∧ (B∨¬C)
α = "A or B" → A∨B

Output:

A	B	C	KB Result	Alpha Result
F	F	F	F	F
F	F	T	F	F
F	T	F	F	T
F	T	T	T	T
T	F	F	T	T
T	F	T	F	T
T	T	F	T	T
T	T	T	T	T



Consider S ∈ T as variables

and following relation →

a : ¬(S ∨ T)
b : (S ∧ T)
c : T ∨ ¬T

Write Truth table and show
i) a entails b
ii) a entails c

(i)

S	T	~(S ∨ T)	KB	α(S ∧ T)
F	F	T	T	F ✗
F	T	F	F	F
T	F	F	F	F
T	T	F	F	T

Knowledge base does not entail α KB ⊭ α

(ii)

S	T	¬(S ∨ T)	KB	T ∨ ¬T
F	F	T	T	T ✓
F	T	F	F	T
T	F	F	F	T
T	T	F	F	T

KB entails α KB ⊨ α

Code:

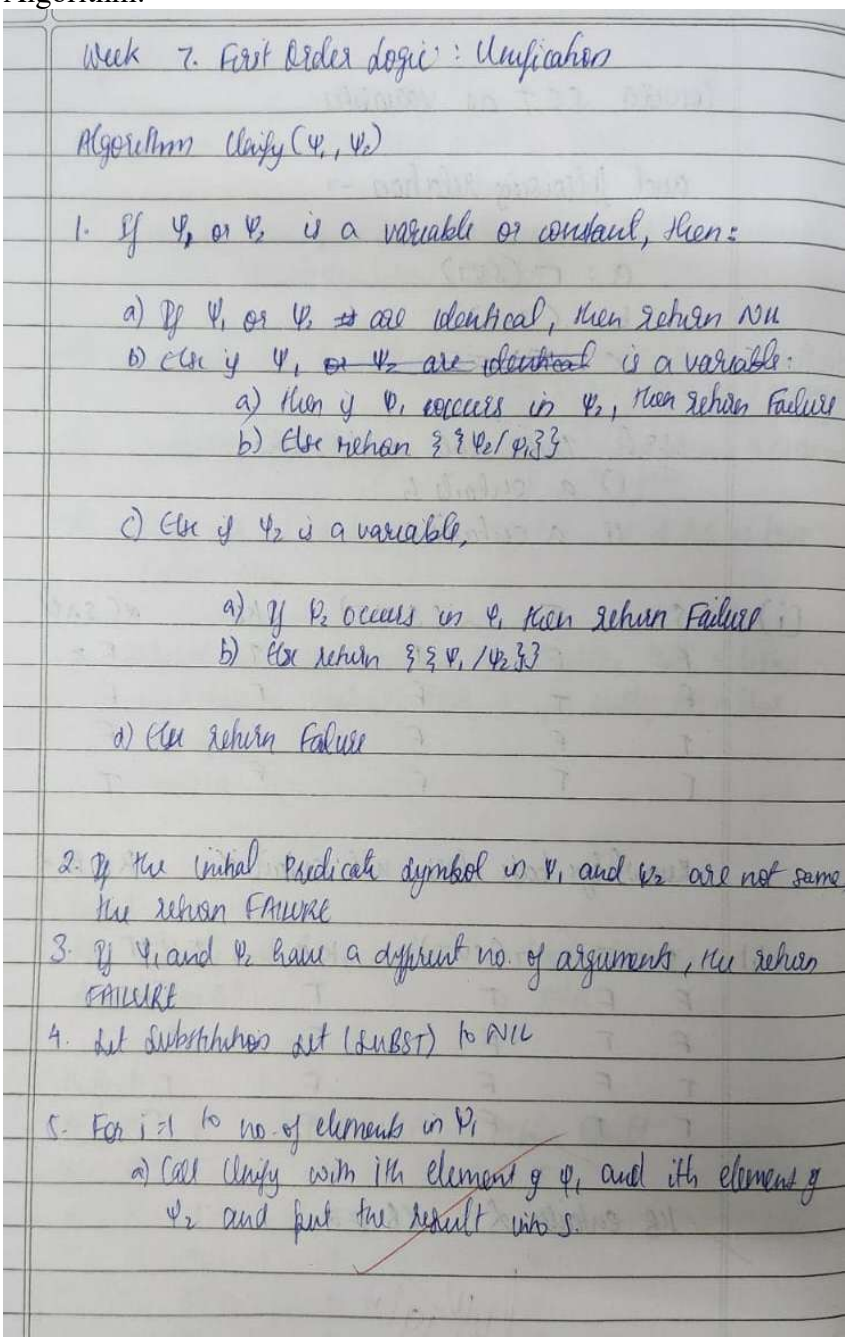
```
import itertools
print("Solution By Sam Manu Jacob - 1BM23CS291")
```

```
def evaluate_formula(formula, truth_assignment):
    eval_formula = formula
    for symbol, value in truth_assignment.items():
        eval_formula = eval_formula.replace(symbol, str(value))
    return eval(eval_formula)

def generate_truth_table(variables):
    return list(itertools.product([False, True], repeat=len(variables)))

def is_entailed(KB_formula, alpha_formula, variables):
    truth_combinations = generate_truth_table(variables)
    print(f"{' '.join(variables)} | KB Result | Alpha Result")
    print("-" * (len(variables) * 2 + 15))
    for combination in truth_combinations:
        truth_assignment = dict(zip(variables, combination))
        KB_value = evaluate_formula(KB_formula, truth_assignment)
        alpha_value = evaluate_formula(alpha_formula, truth_assignment)
        result_str = " ".join(["T" if value else "F" for value in combination])
        print(f"{result_str} | {'T' if KB_value else 'F'}         | {'T' if alpha_value else 'F'}")
        if KB_value and not alpha_value:
            return False
    return True

KB = "(A or C) and (B or not C)"
alpha = "A or B"
variables = ['A', 'B', 'C']

if is_entailed(KB, alpha, variables):
    print("\nThe knowledge base entails alpha.")
else:
    print("\nThe knowledge base does not entail alpha.")
```

Program 7

Implement unification in first order logic

Algorithm:



Week 7. First Order logic : Unification

Algorithm Unify(Ψ_1, Ψ_2)

1. 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 ~~or Ψ_2 are identical~~ is a variable:

      a) then if Ψ_1 occurs in Ψ_2, then return Failure

      b) Else return {Ψ_2/Ψ_1}

   c) Else if Ψ_2 is a variable,

      a) If Ψ_2 occurs in Ψ_1 then return Failure

      b) Else return {Ψ_1/Ψ_2}

   d) Else return Failure

2. If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same then return FAILURE

3. If Ψ_1 and Ψ_2 have a different no. of arguments, then return FAILURE

4. Set Substitution set (SUBST) to NIL

5. For i = 1 to no. of elements in Ψ_1

   a) Call Unify with ith element of Ψ_1 and ith element of Ψ_2 and put the result into S.

PAGE NO :
DATE :

b) If S = failure then return FAILURE
c) If S ≠ NIL then do,
   a. Apply S to the remainder of both L1 and L2
   b. SUBST = APPEND(S, SUBST)

6. Return SUBST

Algorithm

1. Unify {b(b, x, f(g(z))), b(z, f(y), f(y))}

   Compare:

   b = z → z = b

   x = f(y) → x = f(y)

   f(g(z)) = f(y) → y = g(z)

   Substitute ~~x = f(y)~~ ~~into~~ y = g(z) into x = f(y) =
   x = f(g(z)). No contradiction

   MGU = {z/b, ~~x/f(g(z))~~ x/f(g(z)), y/g(z)}

2. Unify Q(a, g(x,a), f(y)) and Q(a, g(f(b), a)), x)

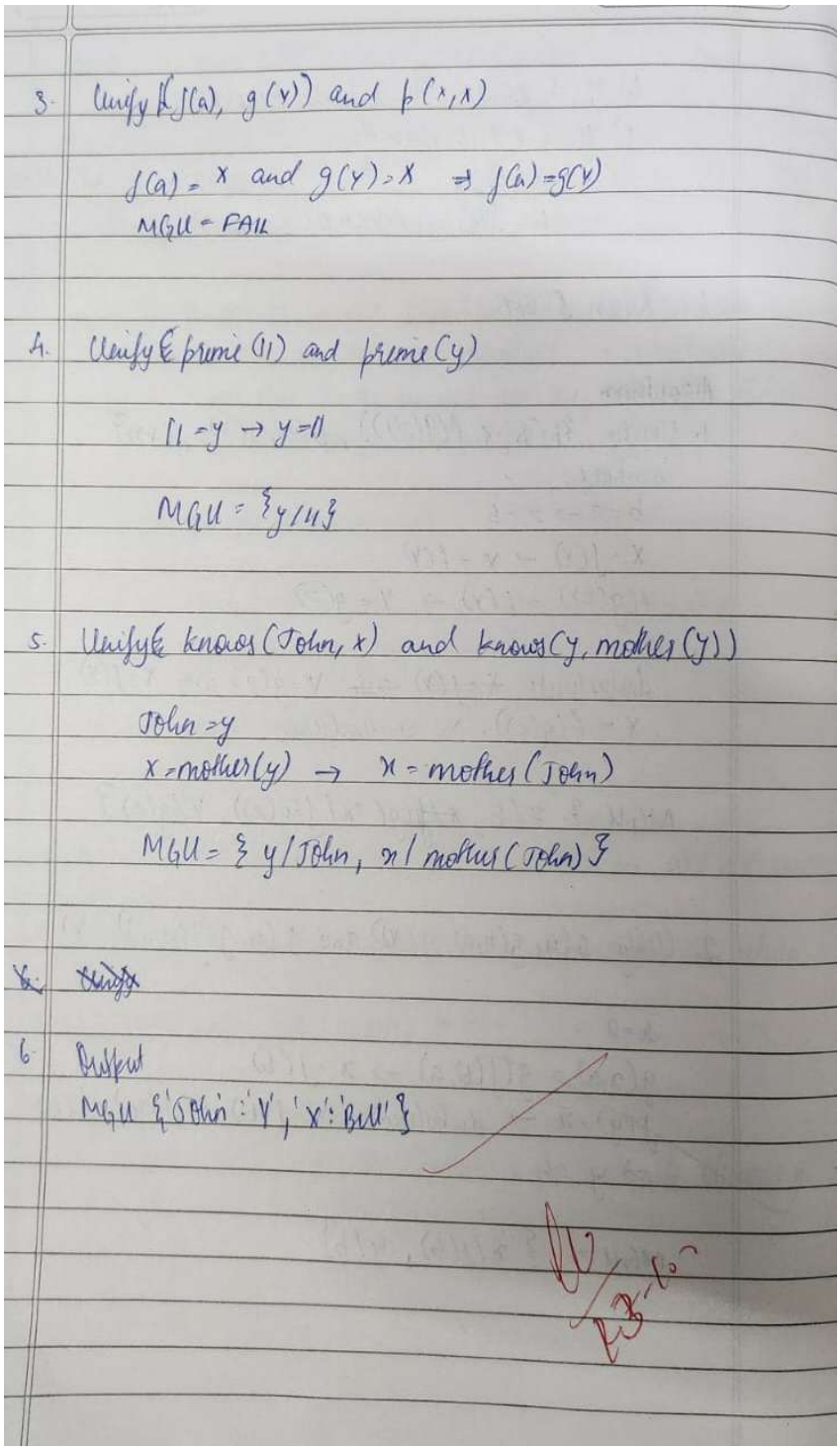
   a = a

   g(x,a) = g(f(b), a) → x = f(b)

   f(y) = x → substitute x = f(b) → f(y) = f(b)

   ⇒ y = b

   MGU = {x/f(b), y/b}



Code:

```
def is_variable(x):
    """Variables are strings that start with an uppercase letter."""
    return isinstance(x, str) and x and x[0].isupper()

def is_compound(x):
    """Compound terms are lists: [functor, arg1, arg2, ...]"""
```

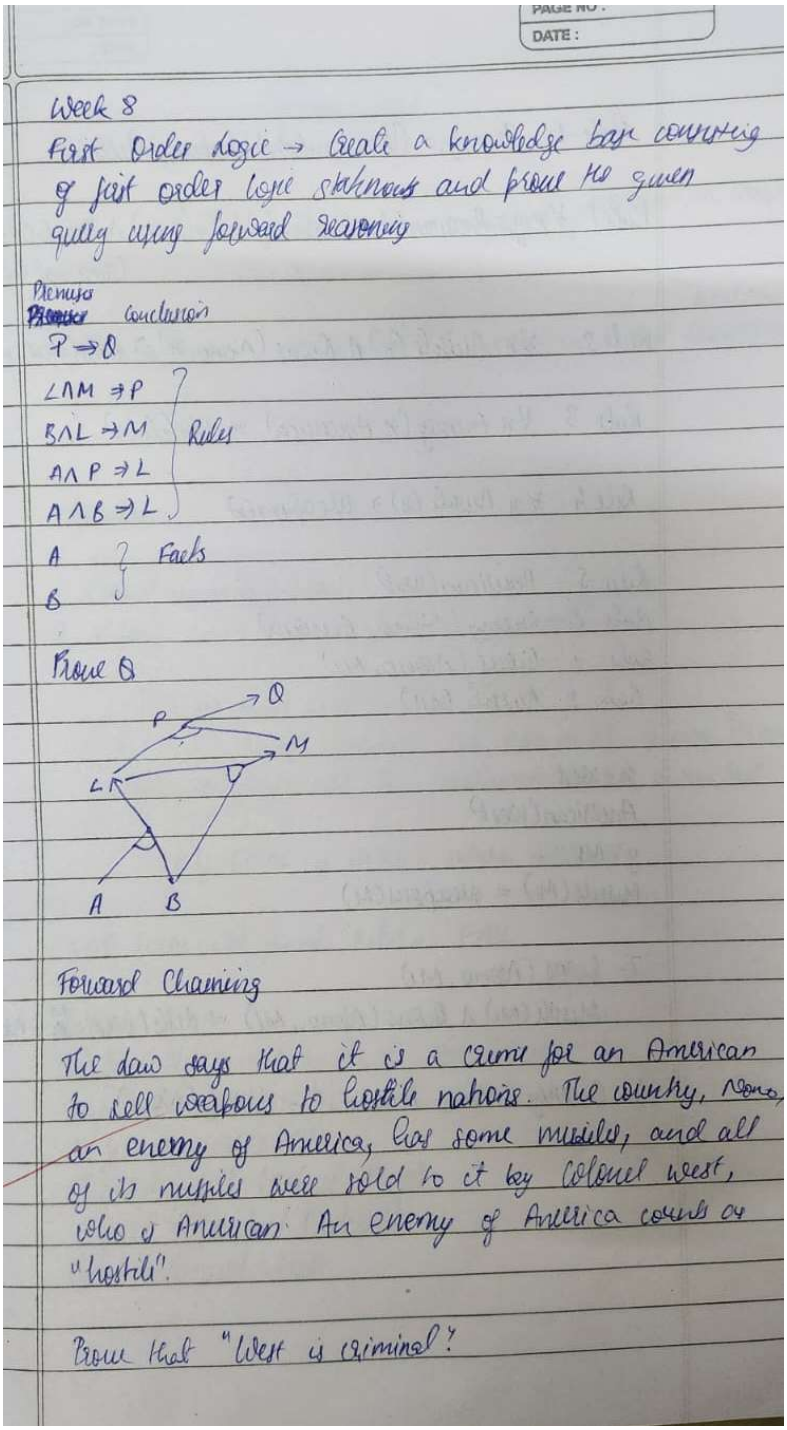
```
    return isinstance(x, list) and len(x) >= 1

def occurs_check(var, x, subst):
    """Return True if var occurs in x."""
    if var == x:
        return True
    if is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    if is_compound(x):
        return any(occurs_check(var, xi, subst) for xi in x[1:])
    return False

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    if is_variable(x) and x in subst:
        return unify(var, subst[x], subst)
    if occurs_check(var, x, subst):
        return "FAIL"
    # otherwise record substitution var -> x (apply existing subst to x)
    subst[var] = substitute(x, subst)
    return subst

def substitute(term, subst):
    """Apply substitution dict to a term (deep)."""
    if is_variable(term):
        return substitute(subst.get(term, term), subst) if term in subst else term
    if is_compound(term):
        return [term[0]] + [substitute(arg, subst) for arg in term[1:]]
    return term  # constant (string)

def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    # apply current substitution before proceeding
    x = substitute(x, subst)
    y = substitute(y, subst)
    if x == y:
        return subst
    if is_variable(x):
        return unify_var(x, y, subst)
    if is_variable(y):
        return unify_var(y, x, subst)
    if is_compound(x) and is_compound(y) and x[0] == y[0] and len(x) == len(y):
        # unify arguments left-to-right
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, subst)
```

```
            if subst == "FAIL":
                return "FAIL"
        return subst
    return "FAIL"



expr1 = ["p", "b", "X", ["f", ["g", "Z"]]]
expr2 = ["p", "Z", ["f", "Y"], ["f", "Y"]]


expr3b = ["knows", "John", "X"]
expr4b = ["knows", "Y", "Bill"]
print("Solution by Sam Manu Jacob-1BM23CS291")
print("=== Problem 1 ===")
res1 = unify(expr1, expr2)
print("MGU:", res1)

print("\n=== Problem 6 ===")
res6 = unify(expr3b, expr4b)
print("MGU:", res6)
```

Program 8

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

Algorithm:



Week 8

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

Premises / Conclusion

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$ } Rules

$A \wedge B \Rightarrow L$

A } Facts

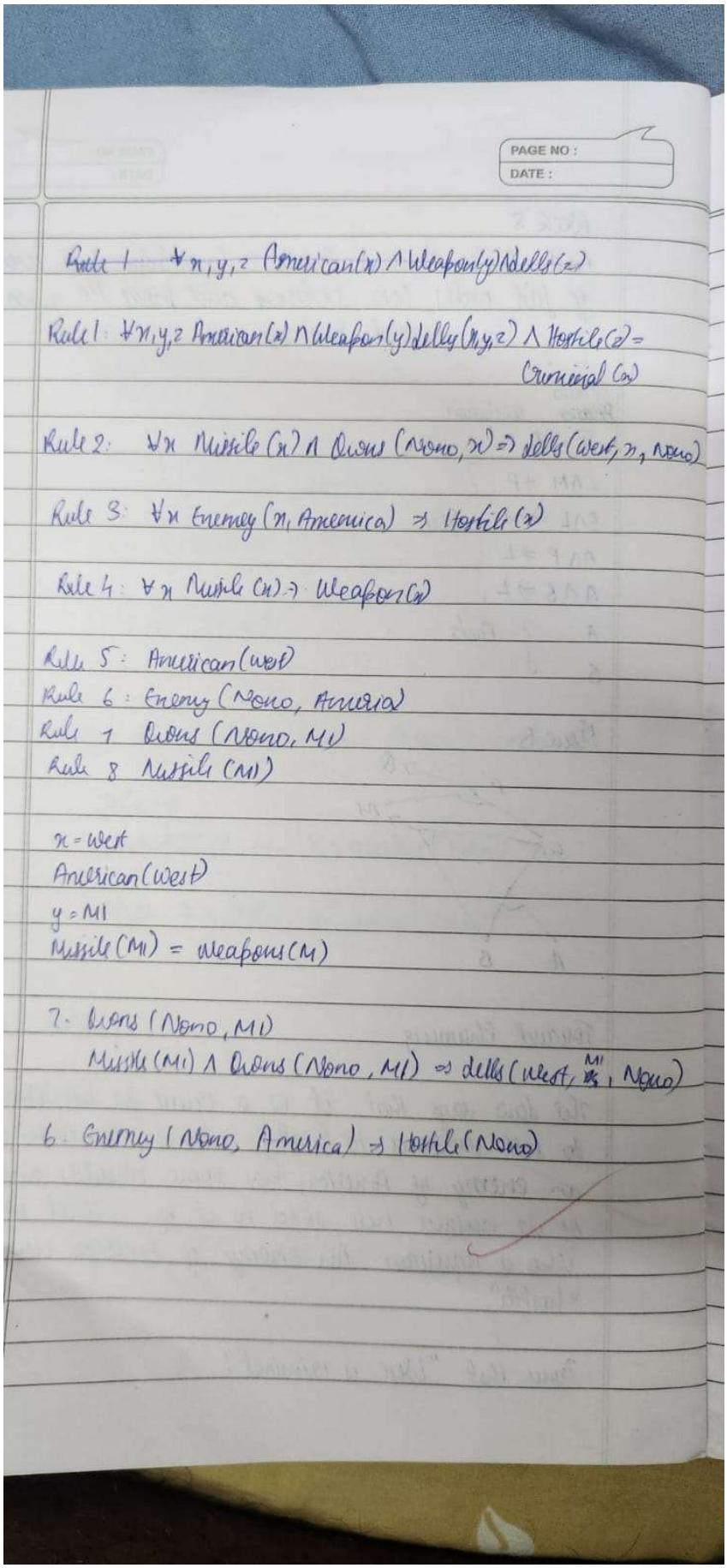
B

Prove Q

Forward Chaining

The law says that it is a crime for an American to sell weapons to hostile nations. The country, Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American. An enemy of America counts as "hostile".

Prove that "West is criminal".



PAGE NO :
DATE :

~~Rule 1~~ $\forall x,y,z$ American(x) $\wedge$ Weapon(y) $\wedge$ Sells(z)

Rule 1: $\forall x,y,z$ American(x) $\wedge$ Weapon(y) $\wedge$ Sells(x,y,z) $\wedge$ Hostile(z) $\Rightarrow$ Criminal(x)

Rule 2: $\forall x$ Missile(x) $\wedge$ Owns(Nono, x) $\Rightarrow$ Sells(West, x, Nono)

Rule 3: $\forall x$ Enemy(x, America) $\Rightarrow$ Hostile(x)

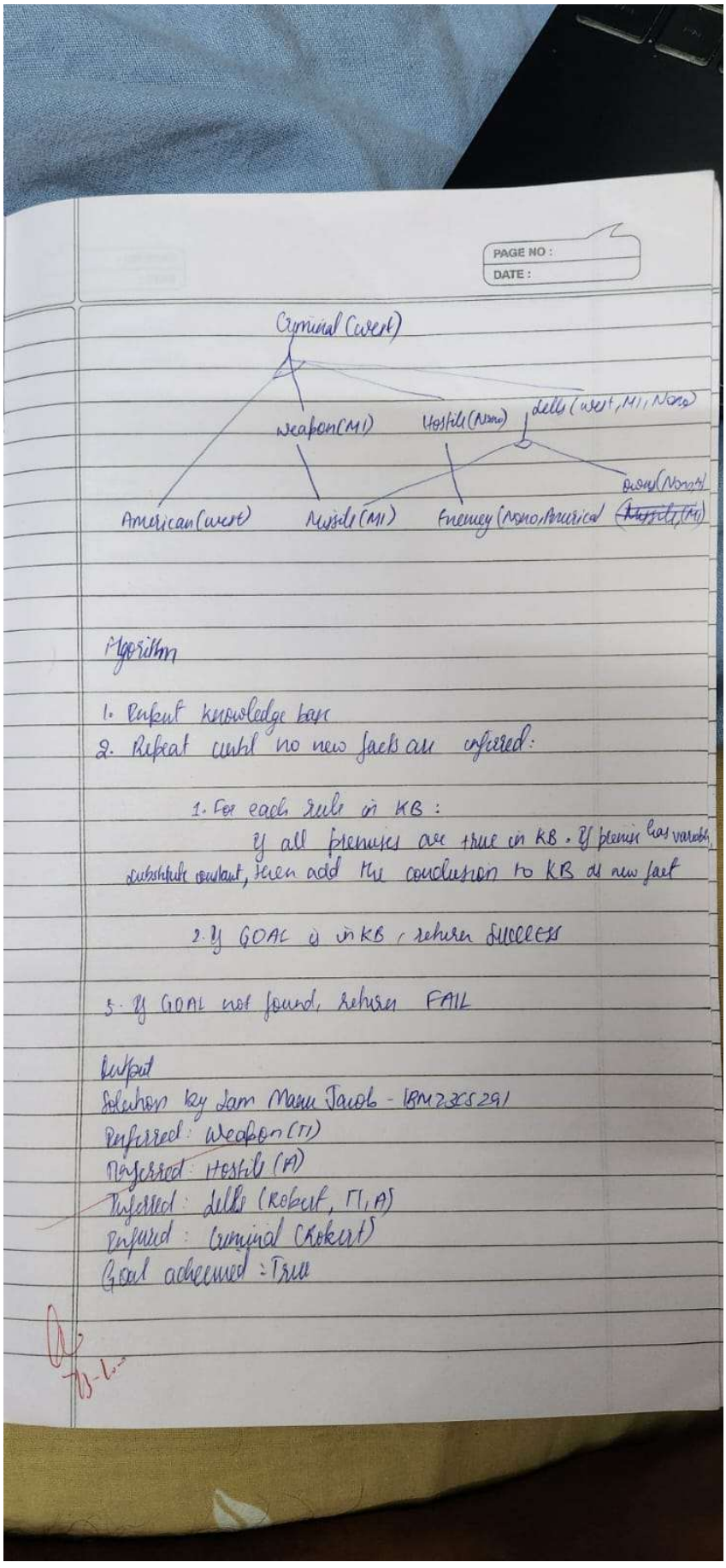
Rule 4: $\forall x$ Missile(x) $\Rightarrow$ Weapon(x)

Rule 5: American(West)
Rule 6: Enemy(Nono, America)
Rule 7: Owns(Nono, M1)
Rule 8: Missile(M1)

x = West
American(West)
y = M1
Missile(M1) $\Rightarrow$ Weapons(M1)

7. Owns(Nono, M1)
Missile(M1) $\wedge$ Owns(Nono, M1) $\Rightarrow$ Sells(West, M1, Nono)

6. Enemy(Nono, America) $\Rightarrow$ Hostile(Nono)



Algorithm

1. Input knowledge base
2. Repeat until no new facts are inferred:

    1. For each rule in KB :
        If all premises are true in KB. If premise has variables substitute constant, then add the conclusion to KB as new fact

    2. If GOAL is in KB, return SUCCESS

    3. If GOAL not found, return FAIL

Output
Solution by Sam Manu Jacob - RM2355291
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Goal achieved: True

Code:

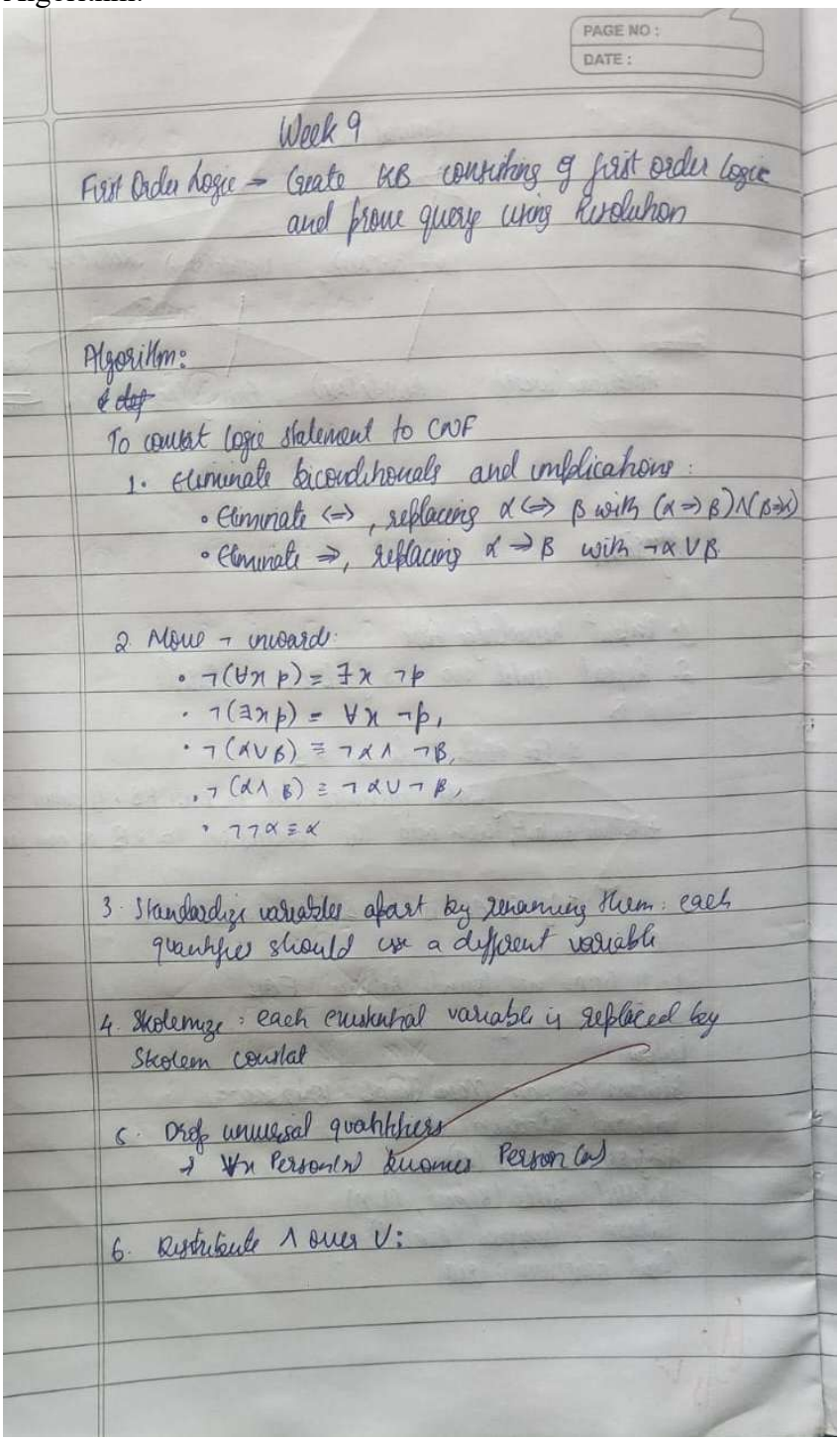
```
import re

print("Solution By Sam Manu Jacob-1BM23CS291")

def match_pattern(pattern, fact):
    """
    Checks if a fact matches a rule pattern using regex-style variable substitution.
    Variables are lowercase words like p, q, x, r etc.
    Returns a dict of substitutions or None if not matched.
    """
    # Extract predicate name and arguments
    pattern_pred, pattern_args = re.match(r'(\w+)', pattern).groups()
    fact_pred, fact_args = re.match(r'(\w+)', fact).groups()

    if pattern_pred != fact_pred:
        return None  # predicate mismatch

    pattern_args = [a.strip() for a in pattern_args.split(",")]
    fact_args = [a.strip() for a in fact_args.split(",")]

    if len(pattern_args) != len(fact_args):
        return None

    subst = {}
    for p_arg, f_arg in zip(pattern_args, fact_args):
        if re.fullmatch(r'[a-z]\w*', p_arg):  # variable
            subst[p_arg] = f_arg
        elif p_arg != f_arg:  # constants mismatch
            return None
    return subst


def apply_substitution(expr, subst):
    """Replaces all variable names in expr using the given substitution dict."""
    for var, val in subst.items():
        expr = re.sub(rf'\b{var}\b', val, expr)
    return expr


# ---------- Knowledge Base ----------

rules = [
    (["American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)"], "Criminal(p)"),
    (["Missile(x)"], "Weapon(x)"),
```

```
    (["Enemy(x, America)"], "Hostile(x)"),
    (["Missile(x)", "Owns(A, x)"], "Sells(Robert, x, A)")
]

facts = {
    "American(Robert)",
    "Enemy(A, America)",
    "Owns(A, T1)",
    "Missile(T1)"
}

goal = "Criminal(Robert)"

def forward_chain(rules, facts, goal):
    added = True
    while added:
        added = False
        for premises, conclusion in rules:

            possible_substs = []
            for p in premises:
                for f in facts:
                    subst = match_pattern(p, f)
                    if subst:
                        possible_substs.append(subst)
                        break
                else:

                    break
            else:

                combined = {}
                for s in possible_substs:
                    combined.update(s)

                new_fact = apply_substitution(conclusion, combined)

                if new_fact not in facts:
                    facts.add(new_fact)
                    print(f"Inferred: {new_fact}")
                    added = True
                    if new_fact == goal:
                        return True
    return goal in facts


print("Goal achieved:", forward_chain(rules, facts, goal)
```

Program 9

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

Algorithm:



Week 9

First Order logic → Create KB consisting of first order logic and prove query using Resolution

Algorithm:

To convert logic statement to CNF

1. Eliminate bicondionals and implications:
   - Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - Eliminate $\Rightarrow$, replacing $\alpha \rightarrow \beta$ with $\neg\alpha \vee \beta$

2. Move $\neg$ inward:
   - $\neg(\forall x\ p) \equiv \exists x\ \neg p$
   - $\neg(\exists x\ p) \equiv \forall x\ \neg p$,
   - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$,
   - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$,
   - $\neg\neg\alpha \equiv \alpha$

3. Standardize variables apart by renaming them: each quantifier should use a different variable

4. Skolemize: each existential variable is replaced by Skolem constant

5. Drop universal quantifiers
   → $\forall x$ Person(x) Enemies Person(a)

6. Distribute $\wedge$ over $\vee$:

PAGE NO :
DATE :

- $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

Resolution:

1. Convert all sentences to CNF
2. Negate conclusion S & 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
   b. Resolve them together
   c. If resolvent is the empty clause, a contradiction has been found
   d. If not, add resolvent to the premises

Output:

Knowledge Base in CNF

1. [¬Food(x), Likes(John, x)]
2. [Food(Apple)]
3. [Food(Vegetable)]
4. [¬Eats(x,y), Killed(x), Food(y)]
5. [Alive(Anil)]
6. [¬Alive(x), ¬Killed(x)]
7. [Killed(x), Alive(x)]

LKW 31/10/25

Negated Query

¬Likes(John, Peanuts)

Empty Clause derived - query proved.

Code:

```
from copy import deepcopy

def print_step(title, content):
    print(f"\n{'='*45}\n{title}\n{'='*45}")
    if isinstance(content, list):
        for i, c in enumerate(content, 1):
            print(f"{i}. {c}")
    else:
        print(content)

KB = [
    ["¬Food(x)", "Likes(John,x)"],
    ["Food(Apple)"],
    ["Food(Vegetable)"],
    ["¬Eats(x,y)", "Killed(x)", "Food(y)"],
    ["Eats(Anil,Peanuts)"],
    ["Alive(Anil)"],
    ["¬Alive(x)", "¬Killed(x)"],
    ["Killed(x)", "Alive(x)"]
]

QUERY = ["Likes(John,Peanuts)"]

def negate(literal):
    if literal.startswith("¬"):
        return literal[1:]
    return "¬" + literal

def substitute(clause, subs):
    new_clause = []
    for lit in clause:
        for var, val in subs.items():
            lit = lit.replace(var, val)
        new_clause.append(lit)
    return new_clause

def unify(lit1, lit2):
    """Small unifier for patterns like Food(x) and Food(Apple)."""
    if "(" not in lit1 or "(" not in lit2:
        return None
    pred1, args1 = lit1.split("(")
    pred2, args2 = lit2.split("(")
    args1 = args1[:-1].split(",")
    args2 = args2[:-1].split(",")
    if pred1 != pred2 or len(args1) != len(args2):
        return None
```

```
    subs = {}
    for a, b in zip(args1, args2):
        if a == b:
            continue
        if a.islower():
            subs[a] = b
        elif b.islower():
            subs[b] = a
        else:
            return None
    return subs

def resolve(ci, cj):
    """Return list of (resolvent, substitution, pair)."""
    resolvents = []
    for li in ci:
        for lj in cj:
            if li == negate(lj):
                new_clause = [x for x in ci if x != li] + [x for x in cj if x != lj]
                resolvents.append((list(set(new_clause)), {}, (li, lj)))
            else:
                # same predicate, opposite sign
                if li.startswith("¬") and not lj.startswith("¬") and li[1:].split("(")[0] == lj.split("(")[0]:
                    subs = unify(li[1:], lj)
                    if subs:
                        new_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs)
                        resolvents.append((list(set(new_clause)), subs, (li, lj)))
                elif lj.startswith("¬") and not li.startswith("¬") and lj[1:].split("(")[0] == li.split("(")[0]:
                    subs = unify(lj[1:], li)
                    if subs:
                        new_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs)
                        resolvents.append((list(set(new_clause)), subs, (li, lj)))
    return resolvents

def resolution(kb, query):
    clauses = deepcopy(kb)
    negated_query = [negate(q) for q in query]
    clauses.append(negated_query)
    print_step("Initial Clauses", clauses)

    steps = []
    new = []
    while True:
        pairs = [(clauses[i], clauses[j]) for i in range(len(clauses))
                 for j in range(i + 1, len(clauses))]
        for (ci, cj) in pairs:
            for r, subs, pair in resolve(ci, cj):
```

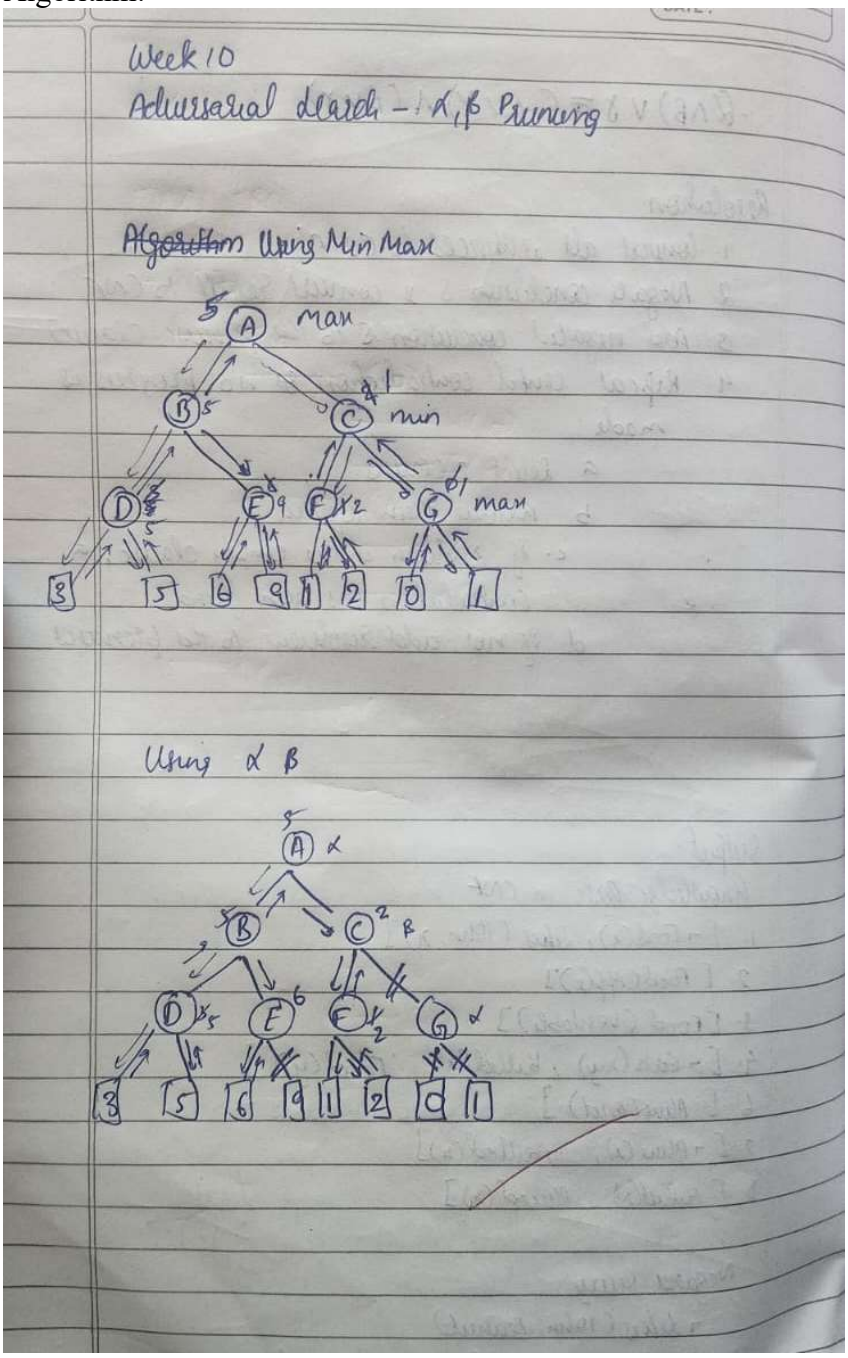
```
            if not r:
                steps.append({
                    "parents": (ci, cj),
                    "resolvent": r,
                    "subs": subs
                })
                print_tree(steps)
                print("\n☑ Empty clause derived — query proven.")
                return True
            if r not in clauses and r not in new:
                new.append(r)
                steps.append({
                    "parents": (ci, cj),
                    "resolvent": r,
                    "subs": subs
                })
        if all(r in clauses for r in new):
            print_step("No New Clauses", "Query cannot be proven ✗")
            print_tree(steps)
            return False
        clauses.extend(new)

def print_tree(steps):
    print("\n" + "="*45)
    print("Resolution Proof Trace")
    print("="*45)
    for i, s in enumerate(steps, 1):
        p1, p2 = s["parents"]
        r = s["resolvent"]
        subs = s["subs"]
        subs_text = f"  Substitution: {subs}" if subs else ""

        print(f"  Resolve {p1} and {p2}")
        if subs_text:
            print(subs_text)
        if r:
            print(f"  ⇒ {r}")
        else:
            print("  ⇒ {} (empty clause)")
        print("-"*45)

def main():
    print_step("Knowledge Base in CNF", KB)
    print_step("Negated Query", [negate(q) for q in QUERY])
    proven = resolution(KB, QUERY)
    if proven:
        print("\n Query Proven by Resolution: John likes peanuts.")
```

```
    else:
        print("\n Query cannot be proven from KB.")

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

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



DATE :

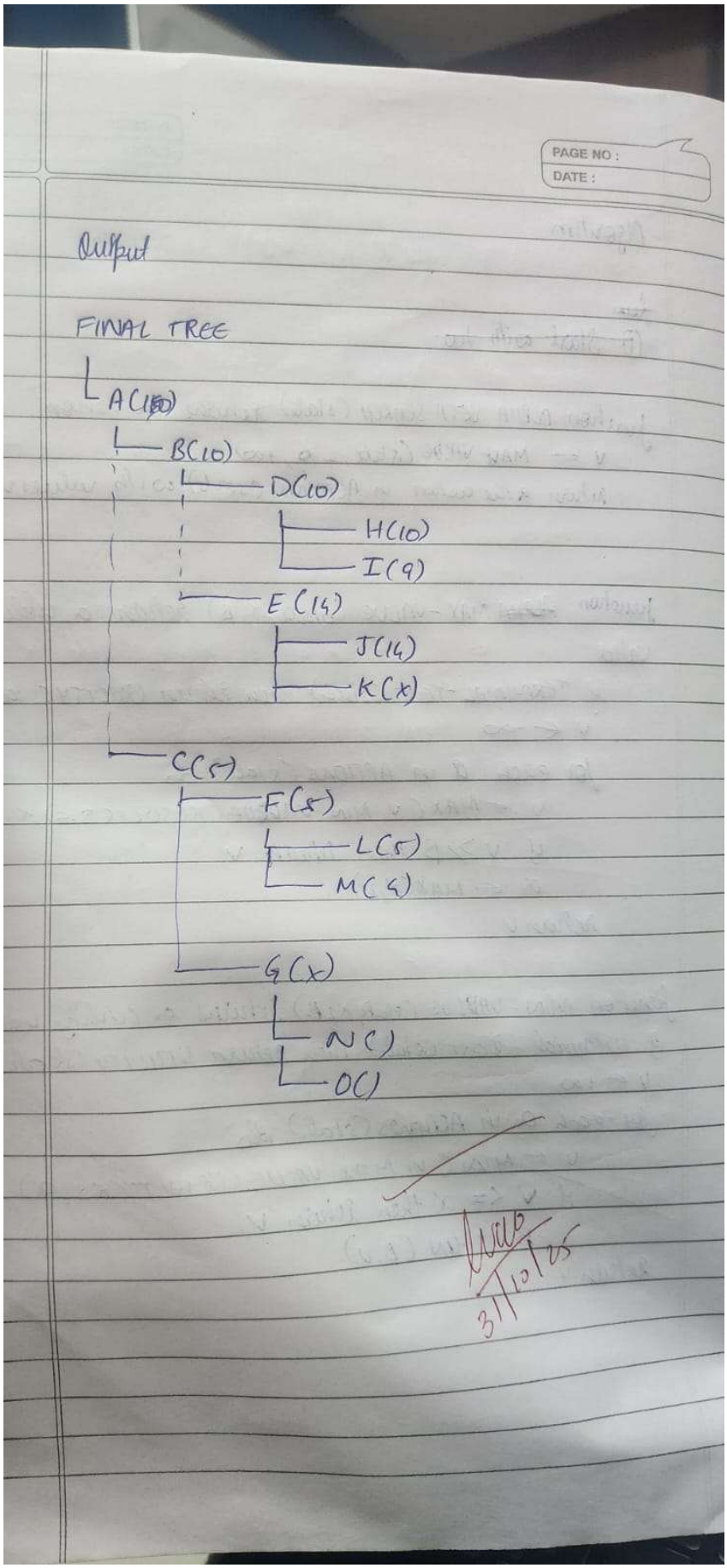
Algorithm

~~for~~
~~① Start with too~~

function ALPHA-BETA-SEARCH (state) returns an action
v ← MAX-VALUE (state, −∞, +∞)
return the action in ACTIONS (state) with value v

function ~~ALPHA~~ MAX-VALUE (state, α, β) returns a utility value
if TERMINAL-TEST (state) then return UTILITY (state)
v ← −∞
for each a in ACTIONS (state) do
v ← MAX (v, MIN-VALUE (RESULT (s, a), α, β))
if v ≥ β then return v
α ← MAX (α, v)
return v

function MIN-VALUE (state, α, β) returns a utility value
if terminal-TEST (state) then return UTILITY (state)
v ← +∞
for each a in ACTIONS (state) do
v ← MIN (v, MAX-VALUE (RESULT (s, a), α, β))
if v <= α then return v
β ← MIN (β, v)
return v


PAGE NO :
DATE :
Output
FINAL TREE
A(
B(10)
D(10)
H(10)
I(9)
E(14)
J(14)
K(x)
C(5)
F(5)
L(5)
M(4)
G(x)
N()
O()
3/12/25

Code:

```
import math

print("Output by Sam Manu Jacob- 1BM23CS291")

tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H', 'I'],
    'E': ['J', 'K'],
    'F': ['L', 'M'],
    'G': ['N', 'O'],
    'H': [], 'I': [], 'J': [], 'K': [],
    'L': [], 'M': [], 'N': [], 'O': []
}

# Leaf node values
values = {
    'H': 10, 'I': 9,
    'J': 14, 'K': 18,
    'L': 5, 'M': 4,
    'N': 50, 'O': 3
}

# to store final display values
node_values = {}

def get_children(node):
    return tree.get(node, [])

def is_terminal(node):
    return len(get_children(node)) == 0

def evaluate(node):
    return values[node]

def alpha_beta(node, depth, alpha, beta, maximizing):
    if is_terminal(node) or depth == 0:
        val = evaluate(node)
        node_values[node] = val
        return val

    if maximizing:
        value = -math.inf
        for child in get_children(node):
            val = alpha_beta(child, depth - 1, alpha, beta, False)
```

```
            value = max(value, val)
            alpha = max(alpha, val)
            if beta <= alpha:
                # mark remaining children as pruned
                for rem in get_children(node)[get_children(node).index(child)+1:]:
                    node_values[rem] = "X"
                break
        node_values[node] = value
        return value
    else:
        value = math.inf
        for child in get_children(node):
            val = alpha_beta(child, depth - 1, alpha, beta, True)
            value = min(value, val)
            beta = min(beta, val)
            if beta <= alpha:
                for rem in get_children(node)[get_children(node).index(child)+1:]:
                    node_values[rem] = "X"
                break
        node_values[node] = value
        return value


# Run pruning
alpha_beta('A', depth=4, alpha=-math.inf, beta=math.inf, maximizing=True)

def print_tree(node, prefix="", is_last=True):
    connector = " └── " if is_last else " ├── "
    value = node_values.get(node, "")
    print(prefix + connector + f"{node} ({value})")
    children = get_children(node)
    for i, child in enumerate(children):
        new_prefix = prefix + ("    " if is_last else " │  ")
        print_tree(child, new_prefix, i == len(children)-1)

# Display the final tree
print("\nFINAL TREE\n" )
print_tree('A')
```