VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Diptanshu Shekhar (1BM23CS361)

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 **Diptanshu Shekhar (1BM23CS361),** 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.

Megha J Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

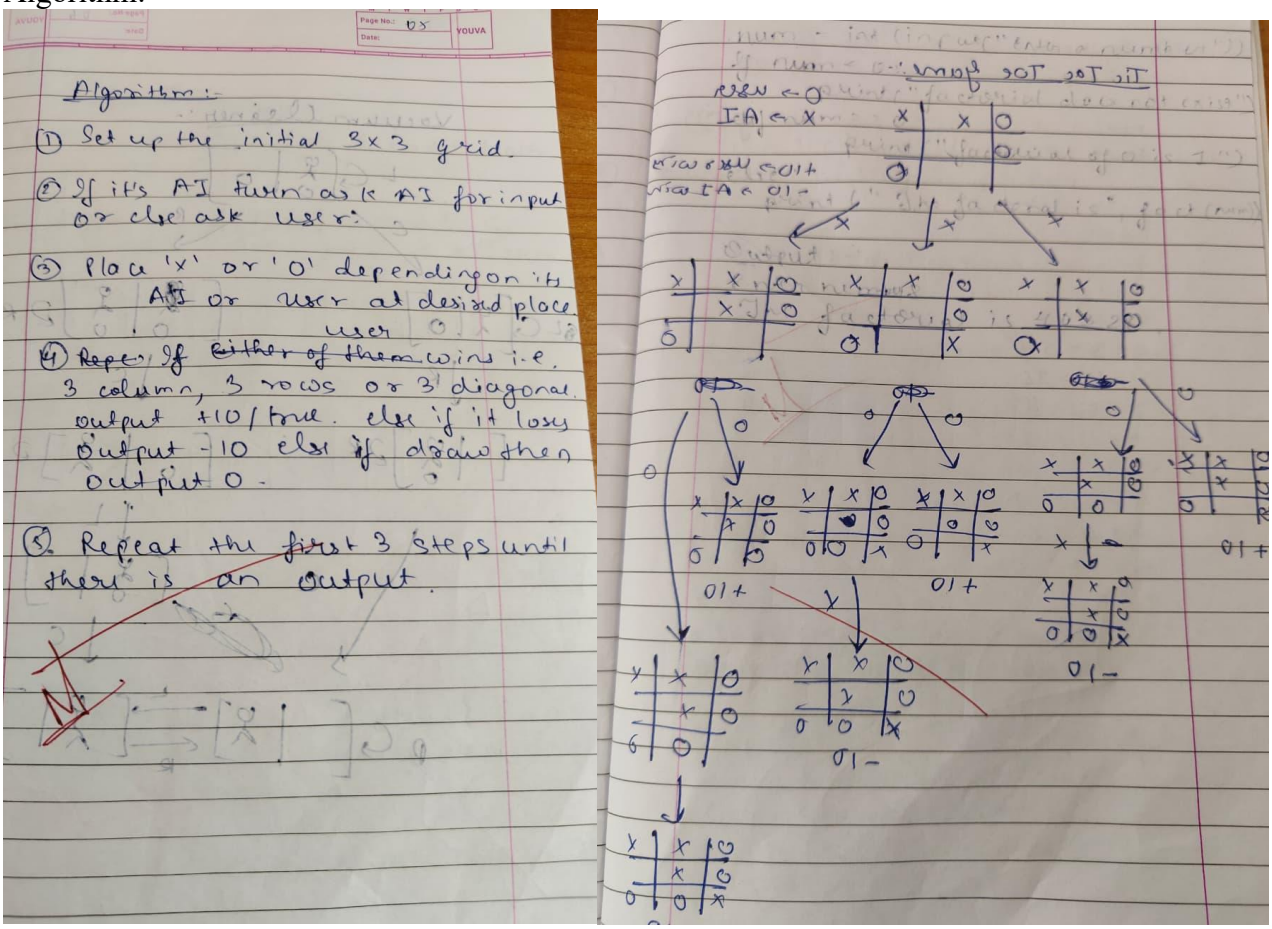
Index

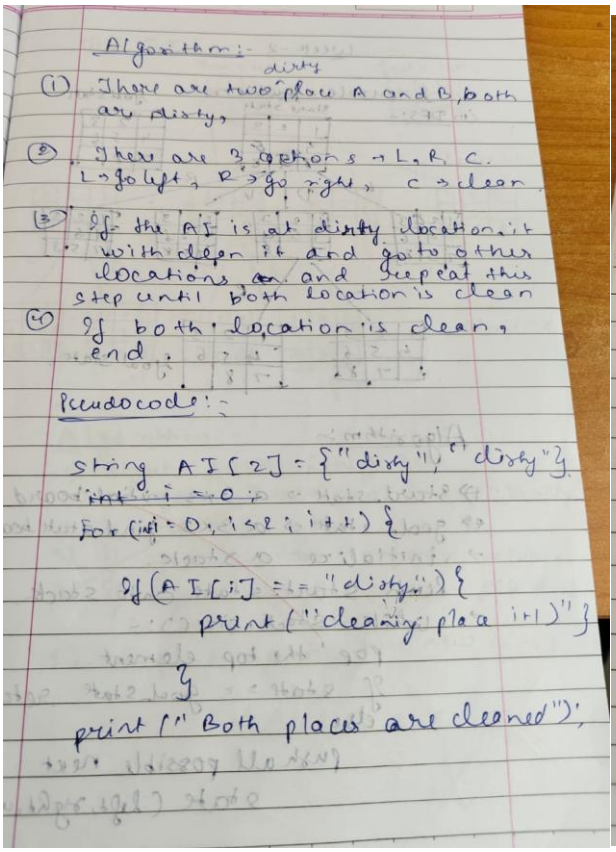
Github Link:
https://github.com/DiptanshuShekharCSE/AI-LAB

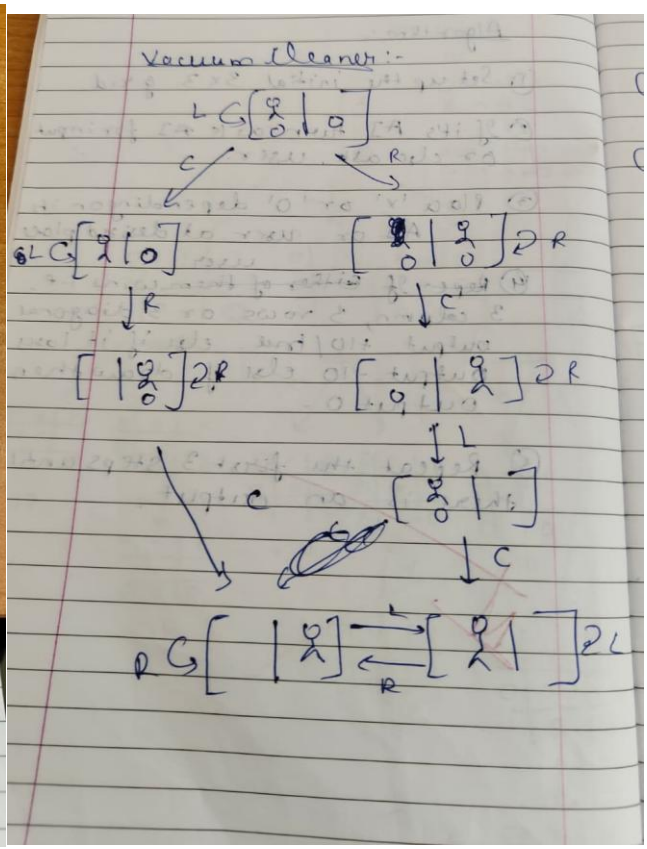
Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:







Code:
a)Tic-Tac-Toe game:

```
def print_board(board):
    for r, row in enumerate(board):
        print(" | ".join(row))
        if r < 2:
            print("-" * 9)

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

def is_full(board):
    return all(cell != " " for row in board for cell in row)
```

```
def ai_first_available_move(board):
    for row in range(3):
        for col in range(3):
            if board[row][col] == " ":
                return (row, col)

def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic-Tac-Toe! You are X, AI is O (first available moves).")
    print_board(board)

    while True:
        print("\nYour turn (X)")
        try:
            row = int(input("Enter row (0, 1, 2): "))
            col = int(input("Enter column (0, 1, 2): "))
        except ValueError:
            print("❌ Invalid input, numbers only!")
            continue

        if (row not in [0,1,2]) or (col not in [0,1,2]) or board[row][col] != " ":
            print("❌ Invalid move, try again.")
            continue

        board[row][col] = "X"
        print_board(board)

        if check_winner(board, "X"):
            print("\nYou win!")
            break
        if is_full(board):
            print("\nIt's a draw!")
            break

        print("\nAI's turn (O)...")
        move = ai_first_available_move(board)
        board[move[0]][move[1]] = "O"
        print_board(board)

        if check_winner(board, "O"):
            print("\n💻 AI wins!")
            break
        if is_full(board):
            print("\n🤝 It's a draw!")
            break
```

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

b) vacuum cleaner:

```
import random
room = {
    "A":random.choice(["clean","Dirty"]),
     "B":random.choice(["clean","Dirty"])
}
position = random.choice(["A","B"])
for _ in range(3):
    print(f"vacuum at room {position},room states:{room}")
    if(room[position]=="Dirty"):
        print(f"cleaning room {position}..")
        room[position]="clean"
    else:
        print(f"room {position} already cleaned")
    if position=="A":
        position="B"
    else:
        position="A"
```

output:

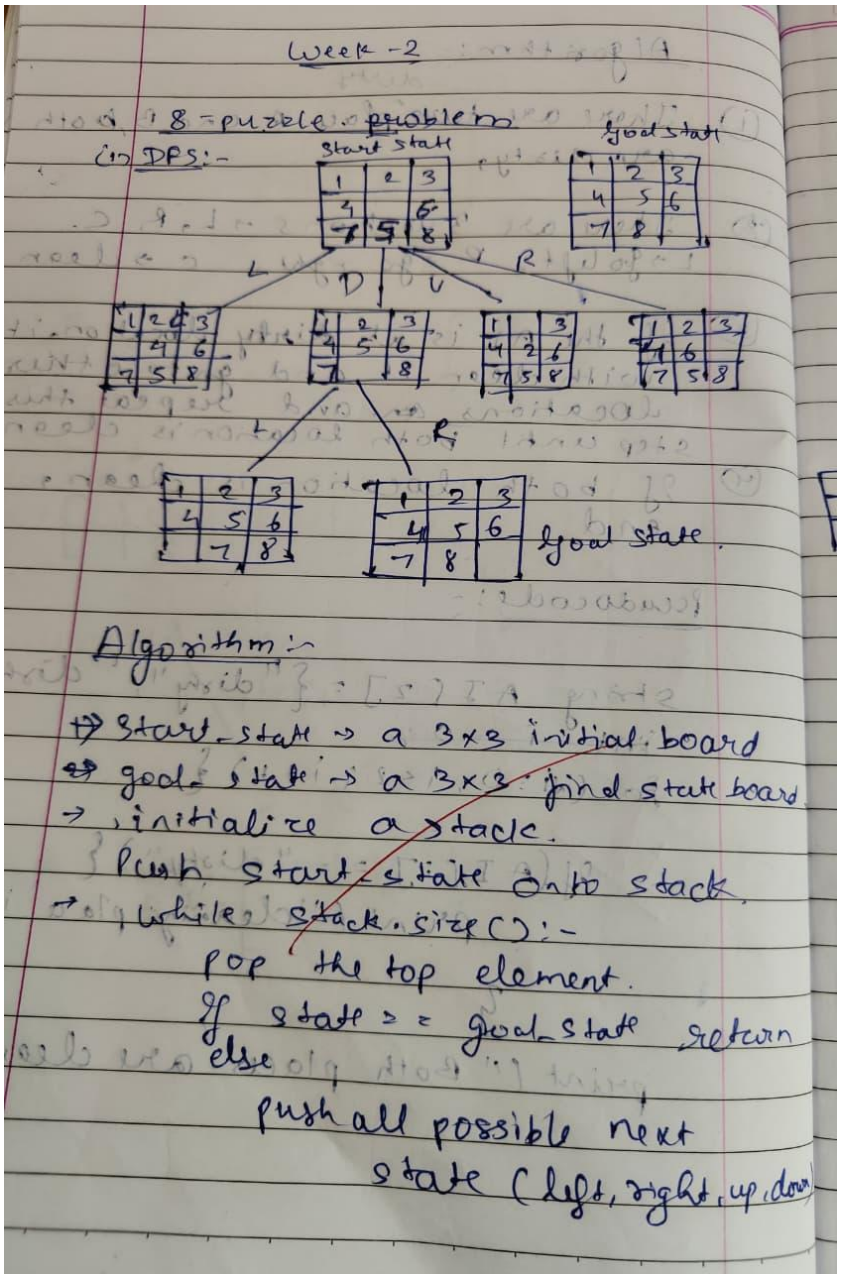
```
vacuum at room B,room states:{'A': 'clean', 'B': 'clean'}
room B already cleaned
vacuum at room A,room states:{'A': 'clean', 'B': 'clean'}
room A already cleaned
vacuum at room B,room states:{'A': 'clean', 'B': 'clean'}
room B already cleaned
```

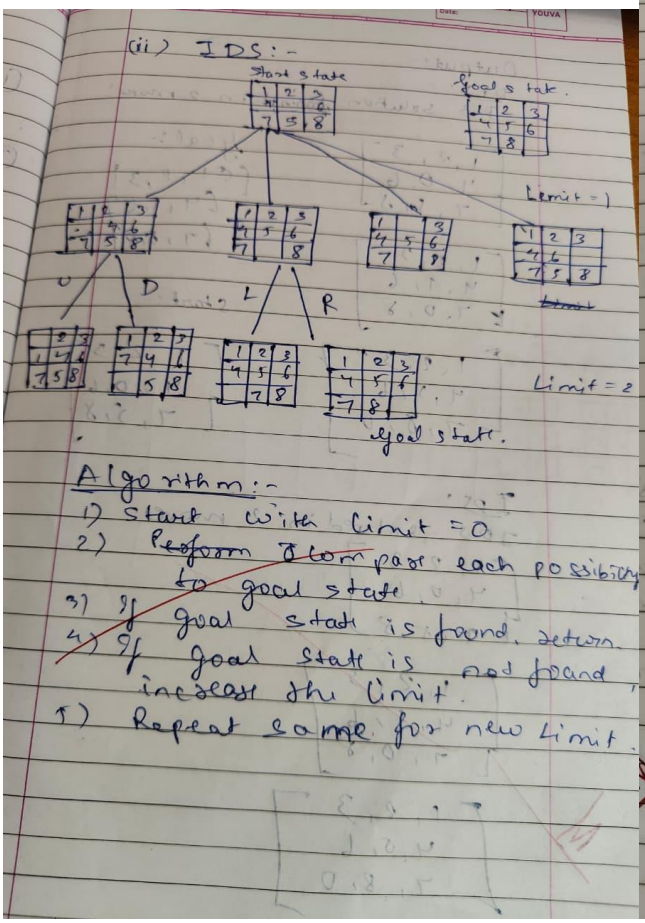
```
Welcome to Tic-Tac-Toe! You are X, AI is O (first available moves).
  |   |
---------
  |   |
---------
  |   |

Your turn (X)
Enter row (0, 1, 2):  0
Enter column (0, 1, 2):  0
X |   |
---------
  |   |
---------
  |   |

AI's turn (O)...
X | O |
---------
  |   |
---------
  |   |

Your turn (X)
Enter row (0, 1, 2):  1
Enter column (0, 1, 2):  1
X | O |
---------
  | X |
---------
  |   |

AI's turn (O)...
X | O | O
---------
  | X |
---------
  |   |

Your turn (X)
Enter row (0, 1, 2):  2
Enter column (0, 1, 2):  2
X | O | O
---------
  | X |
---------
  |   | X

You win!
```

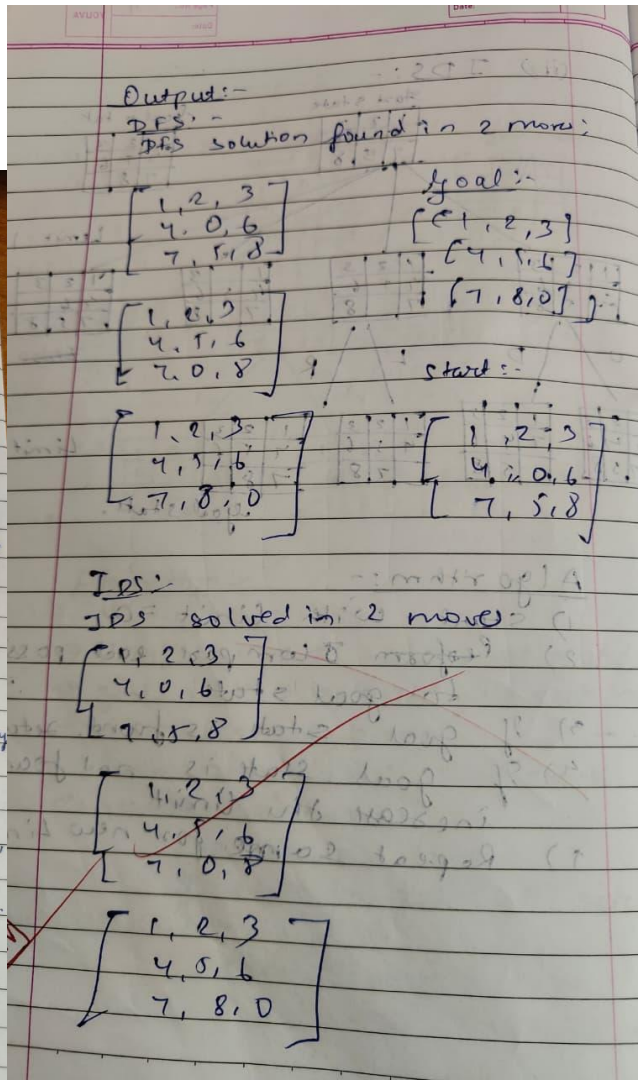
Program 2

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

Algorithm:







Code:

a) from collections import deque

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

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
```

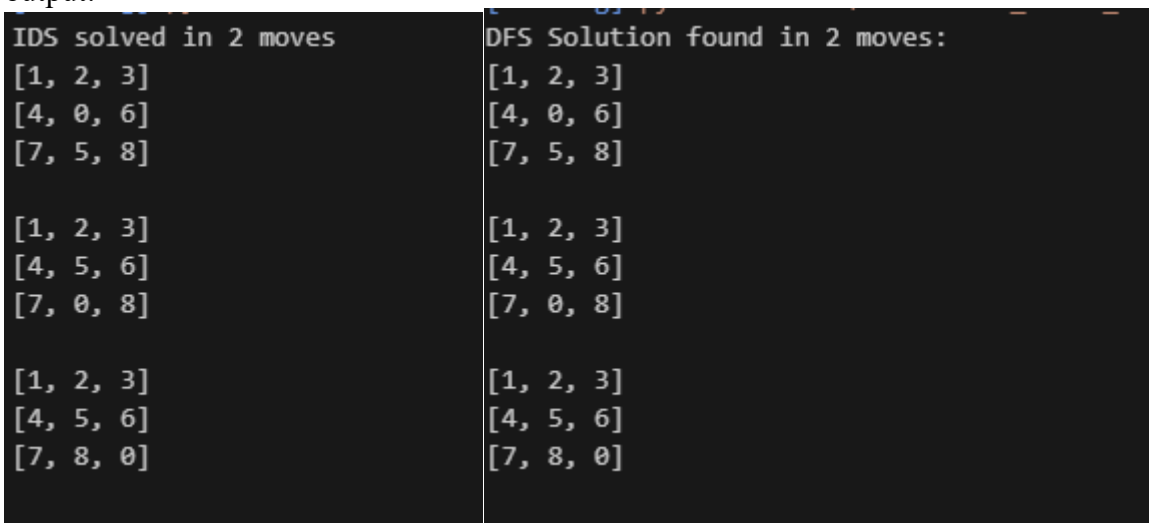
```
def is_goal(state):
    return state == goal_state


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 = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors


def dfs(start_state, max_depth=50):
    stack = [(start_state, [start_state], 0)]
    visited = set()

    while stack:
        state, path, depth = stack.pop()
        state_tuple = tuple(tuple(row) for row in state)

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

        if is_goal(state):
            return path

        if depth < max_depth:
            for neighbor in get_neighbors(state):
                stack.append((neighbor, path + [neighbor], depth + 1))

    return None


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

solution = dfs(start)
if solution:
    print("DFS Solution found in", len(solution) - 1, "moves:")
    for step in solution:
```

```
        for row in step:
            print(row)
        print()
else:
    print("No solution found.")
```

b)
```
goal = [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]]

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 neighbors(state):
    x, y = find_blank(state)
    result = []
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new = [row[:] for row in state]
            new[x][y], new[nx][ny] = new[nx][ny], new[x][y]
            result.append(new)
    return result

def dls(state, path, limit):
    if state == goal:
        return path
    if limit == 0:
        return None
    for nb in neighbors(state):
        if nb not in path:
            res = dls(nb, path + [nb], limit - 1)
            if res: return res
    return None

def ids(start, maxd=20):
    for d in range(maxd+1):
        res = dls(start, [start], d)
        if res: return res
    return None

start = [[1, 2, 3],
```

```
        [4, 0, 6],
        [7, 5, 8]]

sol = ids(start)
if sol:
    print("IDS solved in", len(sol)-1, "moves")
    for s in sol:
        for r in s: print(r)
        print()
else:
    print("No solution")
```

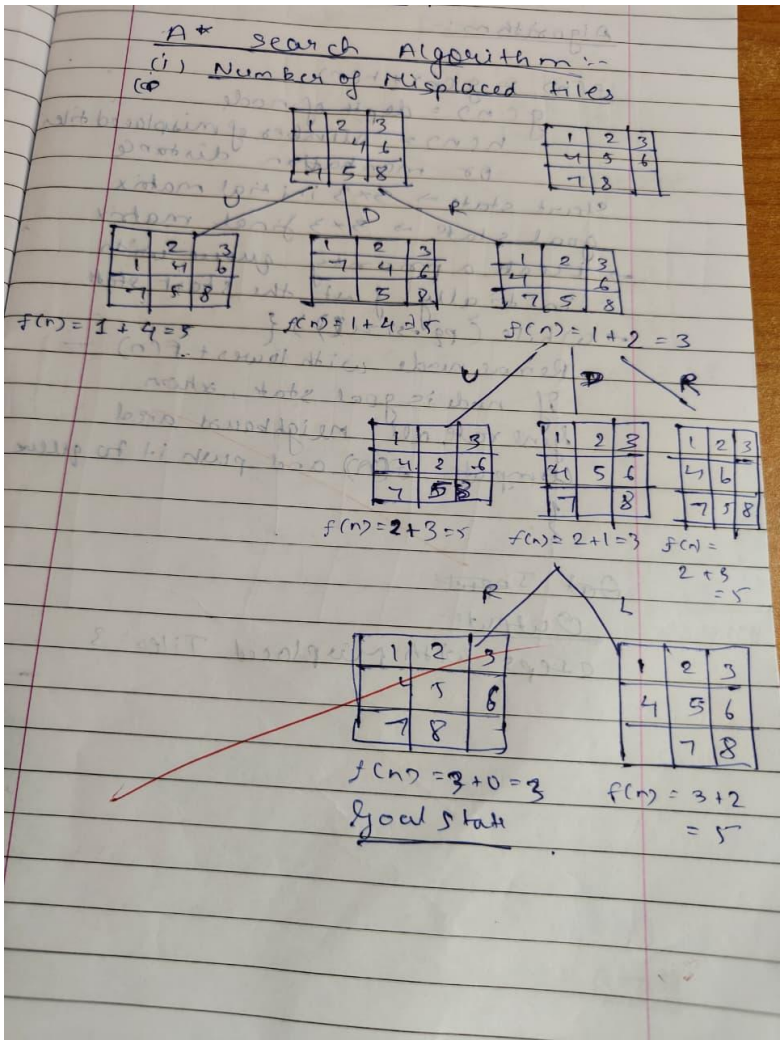
output:

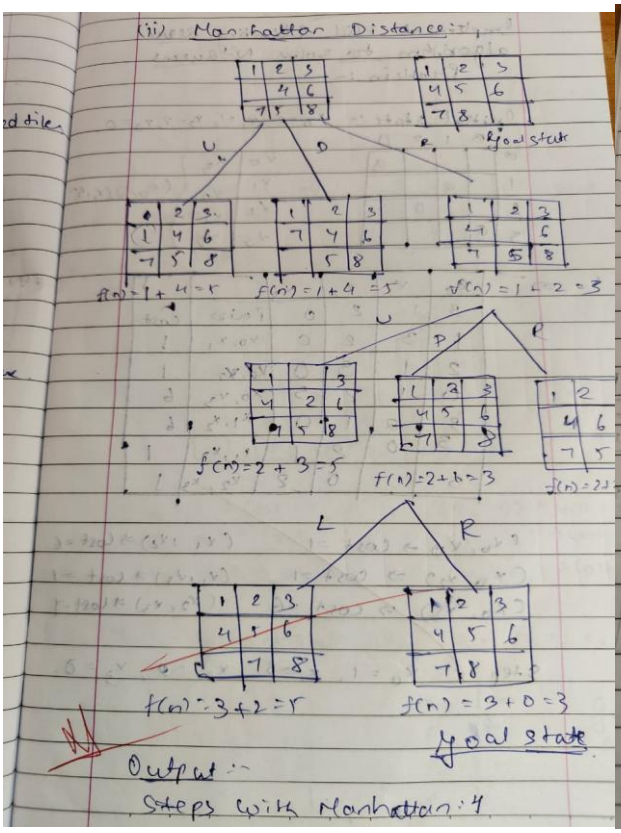


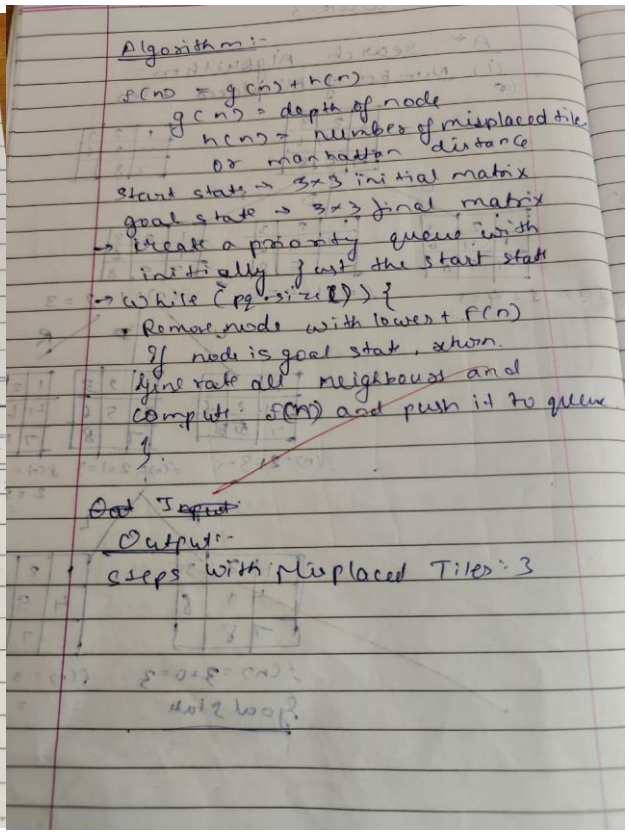
Program 3

Implement A* search algorithm

Algorithm:







Code:
a)misplaced tiles:

```
import heapq

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

def misplaced(state):
    return sum(state[i][j] != 0 and state[i][j] != goal[i][j]
               for i in range(3) for j in range(3))



def to_tuple(state): return tuple(num for row in state for num in row)

def neighbors(state):
    x,y = [(i,j) for i in range(3) for j in range(3) if state[i][j]==0][0]
    dirs = [(1,0),(-1,0),(0,1),(0,-1)]
    for dx,dy in dirs:
        nx,ny = x+dx,y+dy
        if 0<=nx<3 and 0<=ny<3:
            new = [row[:] for row in state]
```

```
            new[x][y], new[nx][ny] = new[nx][ny], new[x][y]
            yield new

def a_star(start, h):
    pq = []
    heapq.heappush(pq,(h(start),0,start))
    seen = {to_tuple(start):0}
    while pq:
        f,g,state = heapq.heappop(pq)
        if state == goal:
            return g
        for nb in neighbors(state):
            ng = g+1
            t = to_tuple(nb)
            if ng < seen.get(t,1e9):
                seen[t] = ng
                heapq.heappush(pq,(ng+h(nb),ng,nb))
    return -1

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

print("Steps with Misplaced Tiles:", a_star(start, misplaced))
```

```
[Running] python -u "d:\1BM23CS36`_Week3_A_MisplaceTiles\MisplacedTiles.py"
Steps with Misplaced Tiles: 3

[Done] exited with code=0 in 0.084 seconds
```

b)Manhattan distance:

```
import heapq

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



def manhattan(state):
    pos = {goal[i][j]:(i,j) for i in range(3) for j in range(3)}
    dist = 0
    for i in range(3):
        for j in range(3):
```

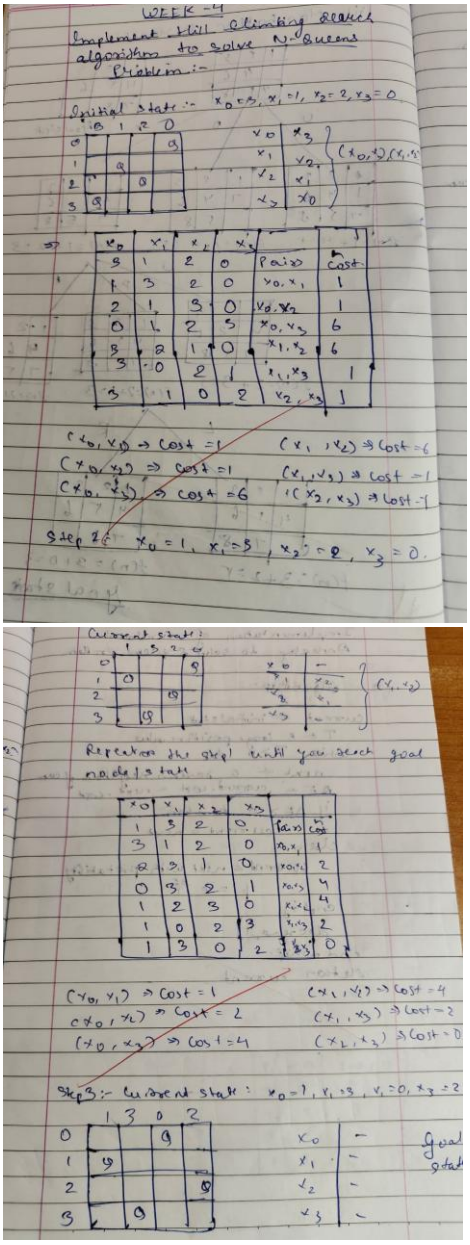
```
            v = state[i][j]
            if v != 0:
                x,y = pos[v]
                dist += abs(x-i) + abs(y-j)
    return dist

def to_tuple(state): return tuple(num for row in state for num in row)

def neighbors(state):
    x,y = [(i,j) for i in range(3) for j in range(3) if state[i][j]==0][0]
    dirs = [(1,0),(-1,0),(0,1),(0,-1)]
    for dx,dy in dirs:
        nx,ny = x+dx,y+dy
        if 0<=nx<3 and 0<=ny<3:
            new = [row[:] for row in state]
            new[x][y], new[nx][ny] = new[nx][ny], new[x][y]
            yield new

def a_star(start, h):
    pq = []
    heapq.heappush(pq,(h(start),0,start))
    seen = {to_tuple(start):0}
    while pq:
        f,g,state = heapq.heappop(pq)
        if state == goal:
            return g
        for nb in neighbors(state):
            ng = g+1
            t = to_tuple(nb)
            if ng < seen.get(t,1e9):
                seen[t] = ng
                heapq.heappush(pq,(ng+h(nb),ng,nb))
    return -1

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


print("Steps with Manhattan:", a_star(start, manhattan))
```

```
[Running] python -u "d:\1BM23CS361_Week3_A_ManhattanDistanceS\Manhattan_Distance.py"
Steps with Manhattan: 4

[Done] exited with code=0 in 0.09 seconds
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



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
    solution, restarts = hill_climb_with_restarts(n)
    print("Found solution:", solution)
    print(format_board(solution))
```

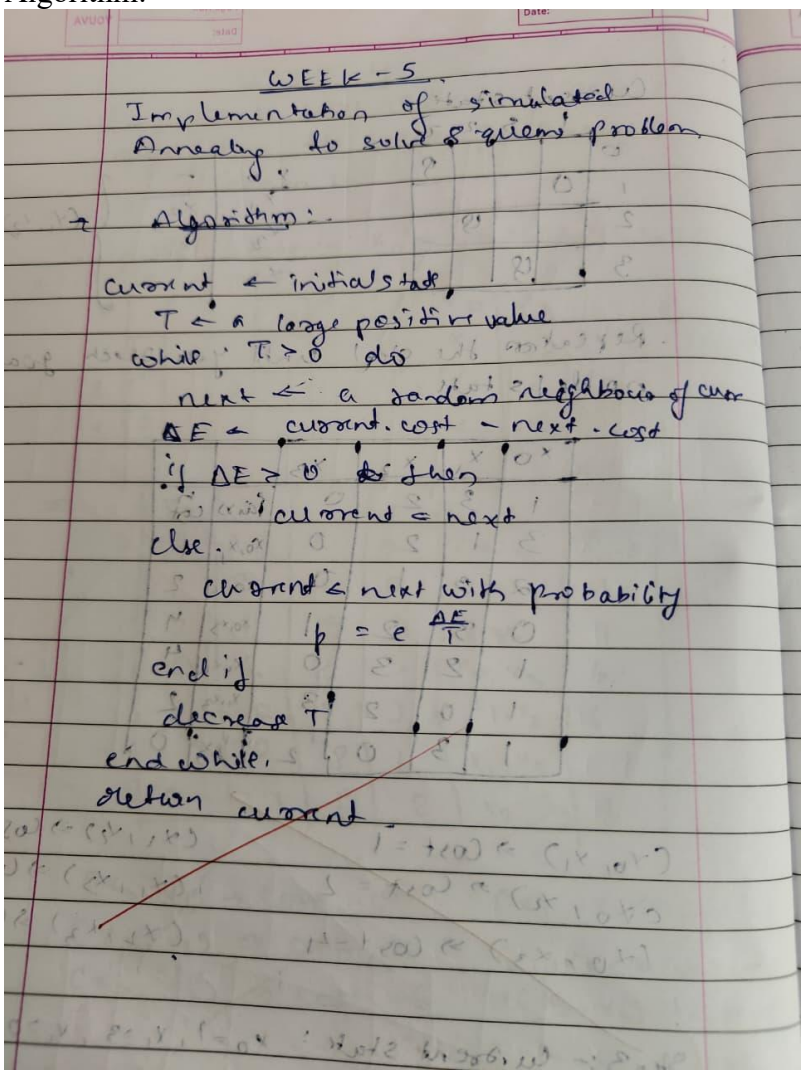
output:

```
Found solution: [2, 0, 3, 1]
- Q - -
- - - Q
Q - - -
- - Q -
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



WEEK - 5

Implementation of simulated Annealing to solve 8 queens problem

Algorithm:

current ← initial state,
T ← a large positive value
while T > 0 do
next ← a random neighbour of current
ΔE ← current.cost − next.cost
if ΔE ≥ 0 then
current ← next
else
current ← next with probability
p = $e^{\frac{\Delta E}{T}}$
end if
decrease T
end while.
return current

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("Best position found:", solution)
print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
print("\nBoard:")
print_board(solution)
```

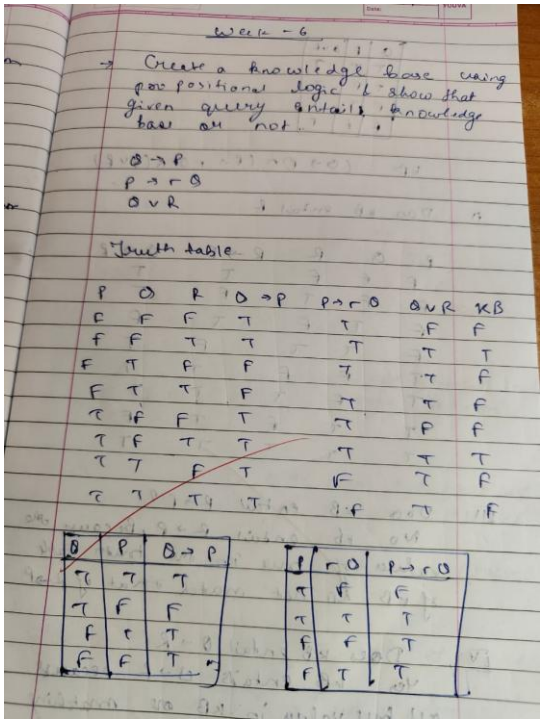
output:

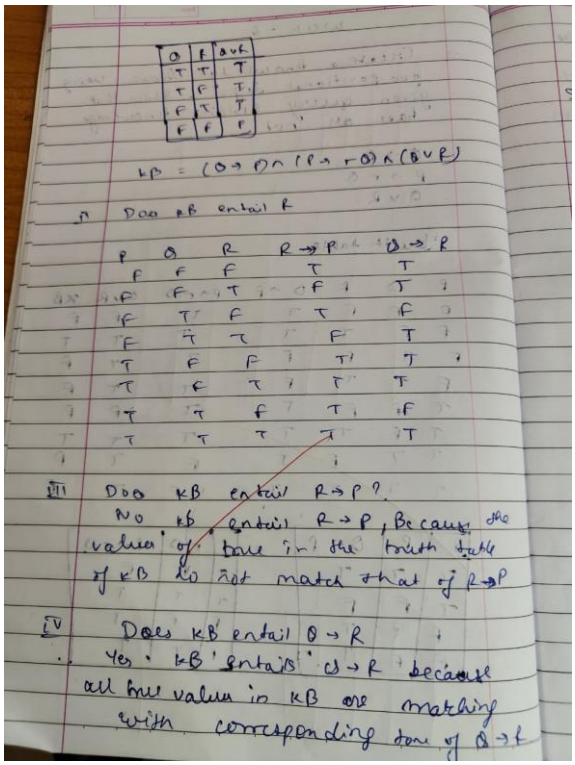
```
Best position found: [2, 5, 7, 0, 4, 6, 1, 3]
Number of non-attacking pairs: 28

Board:
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
```

Program 6

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

Algorithm:





Code:

```
import itertools
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.")
```

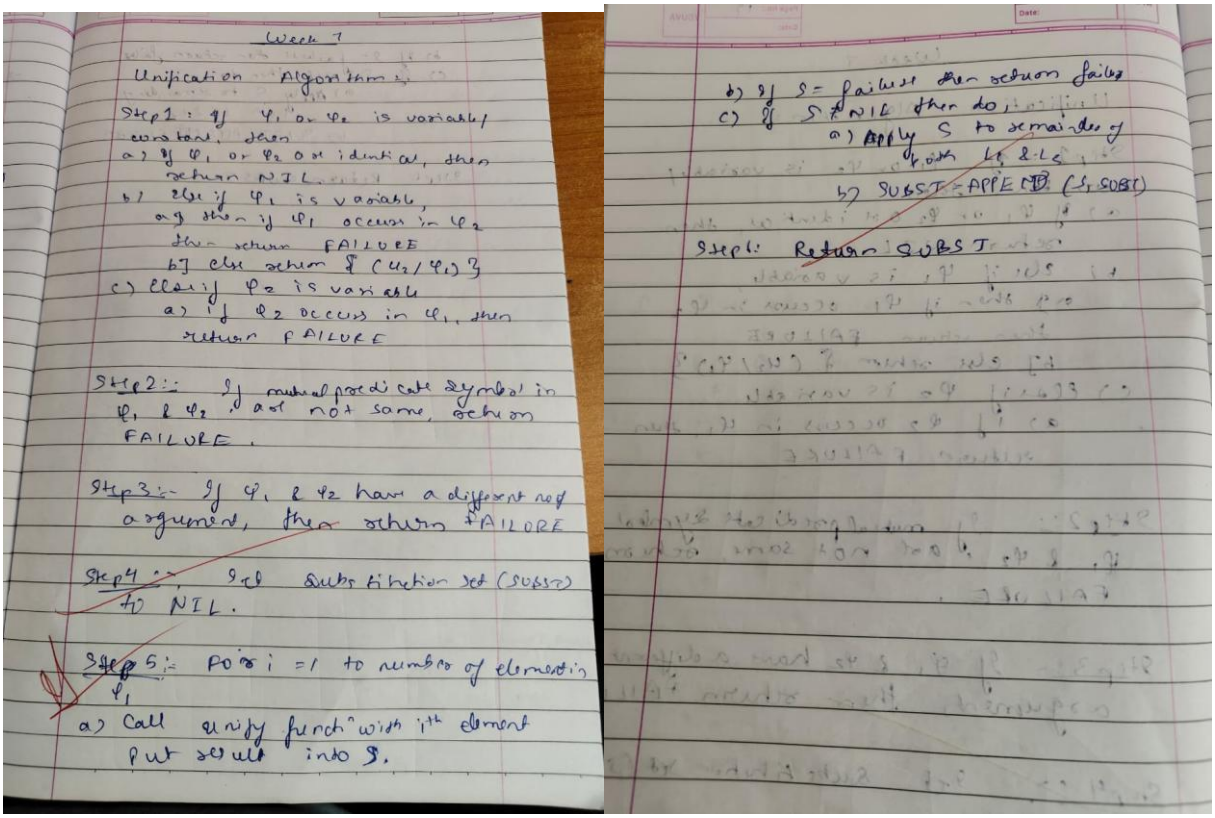
output:

```
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

The knowledge base entails alpha.
```

Program 7

Implement unification in first order logic

Algorithm:



Week 7

Unification Algorithm :-

Step 1 : If ψ_1 or ψ_2 is variable / constant, then

a) If ψ_1 or ψ_2 are identical, then return NIL.

b) else if ψ_1 is variable, and then if ψ_1 occurs in ψ_2 then return FAILURE

b) else return $\{(\psi_2 / \psi_1)\}$

c) else if ψ_2 is variable

a) If ψ_2 occurs in ψ_1, then return FAILURE

Step 2 :- If number of predicate symbol in ψ_1 & ψ_2 are not same, return FAILURE.

Step 3 :- If ψ_1 & ψ_2 have a different no of argument, then return FAILURE

Step 4 :- Set substitution set (SUBST) to NIL.

Step 5 :- For i = 1 to number of elements in ψ_1

a) Call unify func with i^{th} element of ψ_1 & ψ_2 put result into S.

b) If S = failure then return failure

c) If S ≠ NIL then do,

a) Apply S to remainder of both L_1 & L_2

b) SUBST = APPEND (S, SUBST)

Step 6 : Return SUBST

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("=== Problem 1 ===")
res1 = unify(expr1, expr2)
print("MGU:", res1)

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

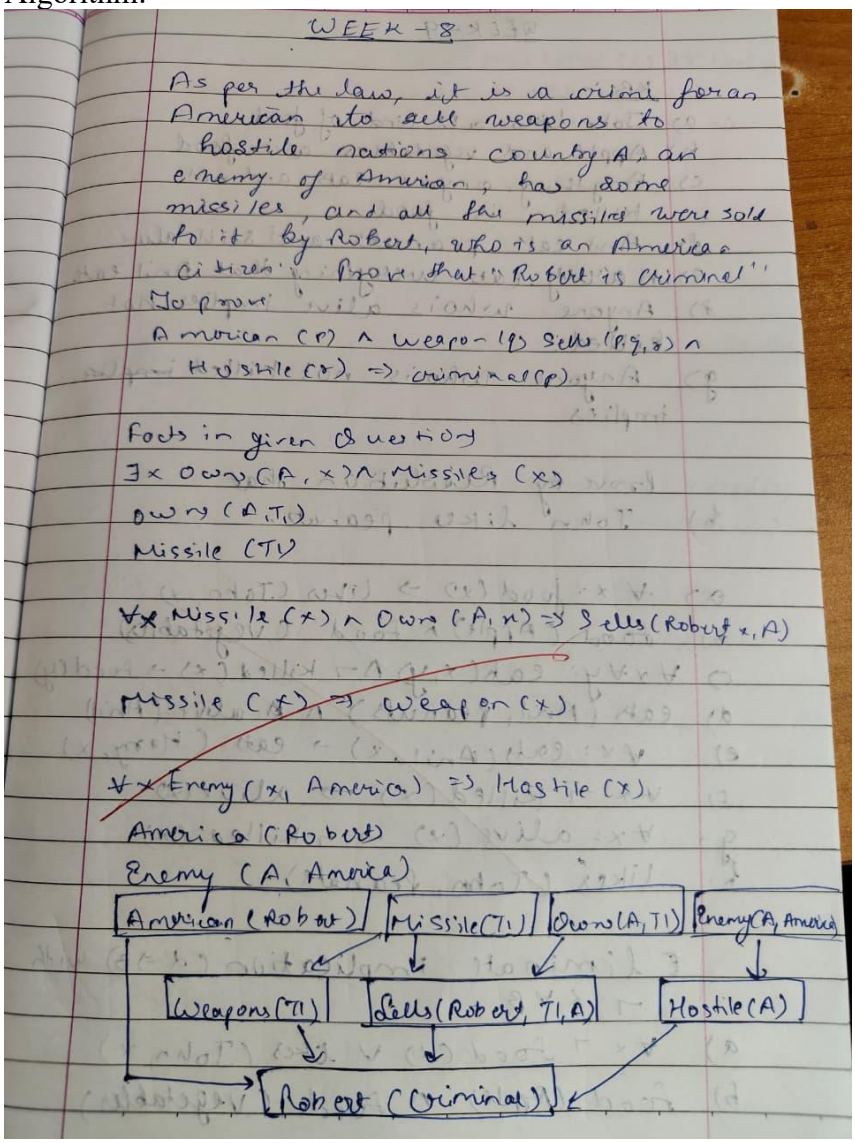
output:

```
=== Problem 1 ===
MGU: {'Z': 'b', 'X': ['f', 'Y'], 'Y': ['g', 'b']}

=== Problem 6 ===
MGU: {'John': 'Y', 'X': 'Bill'}
```

Program 8

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

Algorithm:



Code:

```
import re

facts = set([
    "American(Robert)",
    "Enemy(A,America)",
    "Owns(A,T1)",
    "Missile(T1)"
])
```

```
rules = [
    {
        "if": ["American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)"],
        "then": "Criminal(p)"
    },
    {
        "if": ["Missile(x)", "Owns(A,x)"],
        "then": "Sells(Robert,x,A)"
    },
    {
        "if": ["Missile(x)"],
        "then": "Weapon(x)"
    },
    {
        "if": ["Enemy(x,America)"],
        "then": "Hostile(x)"
    }
]

goal = "Criminal(Robert)"

def match(statement, fact):
    s_pred, s_args = statement.split("(")
    f_pred, f_args = fact.split("(")
    if s_pred != f_pred:
        return None
    s_args = s_args[:-1].split(",")
    f_args = f_args[:-1].split(",")
    if len(s_args) != len(f_args):
        return None
    subs = {}
    for s, f in zip(s_args, f_args):
        if s[0].islower():
            subs[s] = f
        elif s != f:
            return None
    return subs

def substitute(statement, subs):
    for var, val in subs.items():
        statement = re.sub(rf'\b{var}\b', val, statement)
    return statement

def forward_chain(facts, rules, goal):
    new_inferred = True
    while new_inferred:
        new_inferred = False
```

```
        for rule in rules:
            matched_facts = []
            possible_subs = []
            for cond in rule["if"]:
                for fact in facts:
                    subs = match(cond, fact)
                    if subs:
                        matched_facts.append(cond)
                        possible_subs.append(subs)
                        break
            if len(matched_facts) == len(rule["if"]):
                combined = {}
                for d in possible_subs:
                    combined.update(d)
                inferred = substitute(rule["then"], combined)
                if inferred not in facts:
                    print(f"Inferred: {inferred}")
                    facts.add(inferred)
                    new_inferred = True
                    if inferred == goal:
                        print(f"\n☑ Goal {goal} proved by Forward Chaining!")
                        return True
    print(f"\n❌ Goal {goal} cannot be proved with given facts.")
    return False

forward_chain(facts, rules, goal)
```

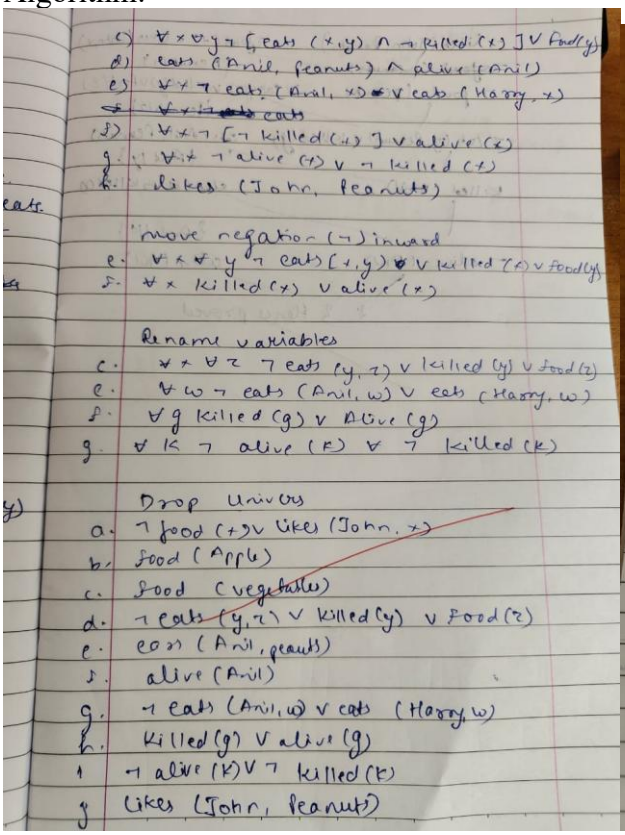
Output:

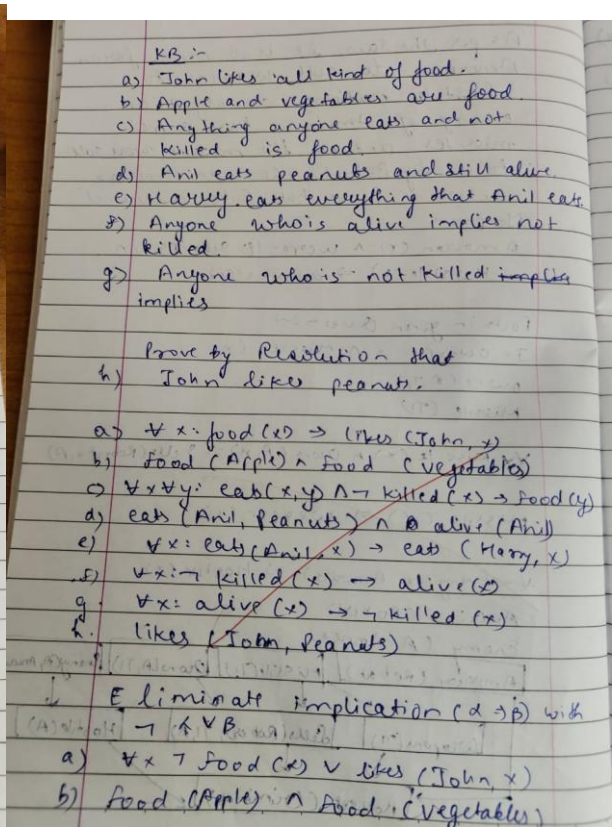
```
Inferred: Sells(Robert,T1,A)
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Criminal(Robert)

✅ Goal Criminal(Robert) proved by Forward Chaining!

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

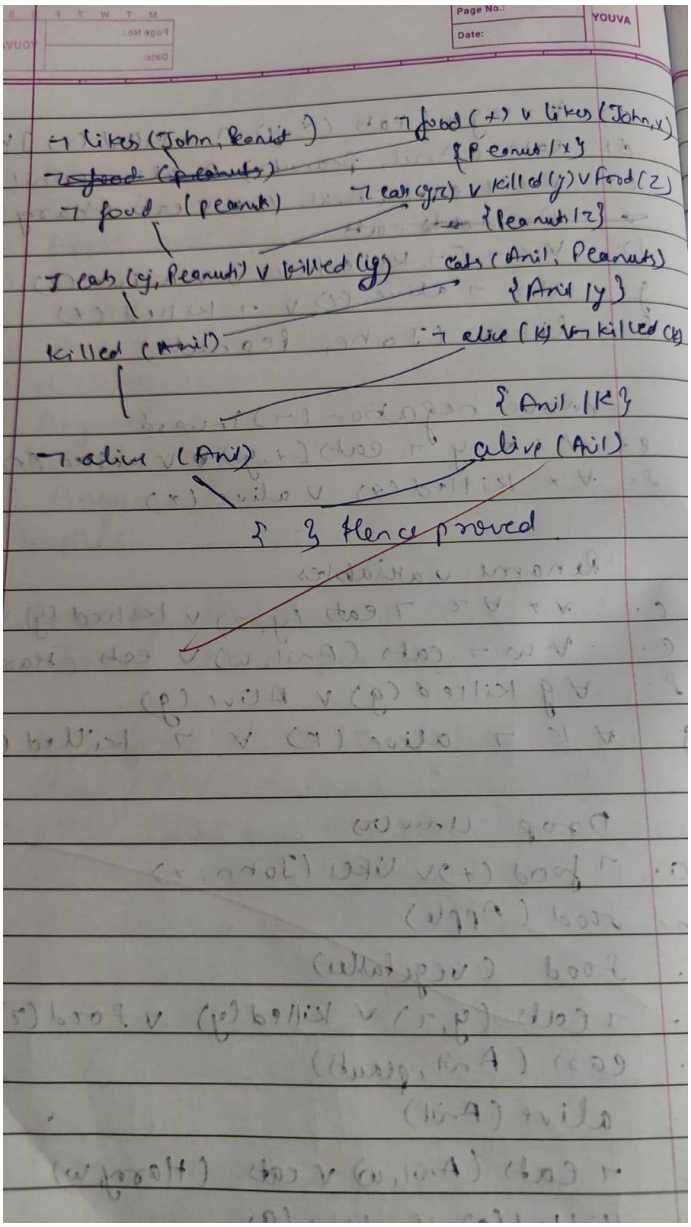
Program 9

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

Algorithm:







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()
```

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. ['Eats(Anil,Peanuts)']
6. ['Alive(Anil)']
7. ['¬Alive(x)', '¬Killed(x)']
8. ['Killed(x)', 'Alive(x)']

==============================================
Negated Query
==============================================
1. ¬Likes(John,Peanuts)

==============================================
Initial Clauses
==============================================
1. ['¬Food(x)', 'Likes(John,x)']
2. ['Food(Apple)']
3. ['Food(Vegetable)']
4. ['¬Eats(x,y)', 'Killed(x)', 'Food(y)']
5. ['Eats(Anil,Peanuts)']
6. ['Alive(Anil)']
7. ['¬Alive(x)', '¬Killed(x)']
8. ['Killed(x)', 'Alive(x)']
9. ['¬Likes(John,Peanuts)']

==============================================
Resolution Proof Trace
==============================================
  Resolve ['¬Food(x)', 'Likes(John,x)'] and ['Food(Apple)']
  Substitution: {'x': 'Apple'}
  ⇒ ['Likes(John,Apple)']
----------------------------------------------
  Resolve ['¬Food(x)', 'Likes(John,x)'] and ['Food(Vegetable)']
  Substitution: {'x': 'Vegetable'}
```

```
  Substitution: {'x': 'y'}
  ⇒ ['Killed(y)', '¬Eats(y,y)', 'Likes(John,y)']
--------------------------------------------
  Resolve ['¬Food(x)', 'Likes(John,x)'] and ['¬Likes(John,Peanuts)']
  Substitution: {'x': 'Peanuts'}
  ⇒ ['¬Food(Peanuts)']
--------------------------------------------
  Resolve ['¬Eats(x,y)', 'Killed(x)', 'Food(y)'] and ['Eats(Anil,Peanuts)']
  Substitution: {'x': 'Anil', 'y': 'Peanuts'}
  ⇒ ['Killed(Anil)', 'Food(Peanuts)']
--------------------------------------------
  Resolve ['¬Eats(x,y)', 'Killed(x)', 'Food(y)'] and ['¬Alive(x)', '¬Killed(x)']
  ⇒ ['Food(y)', '¬Alive(x)', '¬Eats(x,y)']
--------------------------------------------
  Resolve ['Alive(Anil)'] and ['¬Alive(x)', '¬Killed(x)']
  Substitution: {'x': 'Anil'}
  ⇒ ['¬Killed(Anil)']
--------------------------------------------
  Resolve ['¬Alive(x)', '¬Killed(x)'] and ['Killed(x)', 'Alive(x)']
  ⇒ ['Killed(x)', '¬Killed(x)']
--------------------------------------------
  Resolve ['¬Alive(x)', '¬Killed(x)'] and ['Killed(x)', 'Alive(x)']
  ⇒ ['Alive(x)', '¬Alive(x)']
--------------------------------------------
  Resolve ['¬Food(x)', 'Likes(John,x)'] and ['Killed(Anil)', 'Food(Peanuts)']
  Substitution: {'x': 'Peanuts'}
  ⇒ ['Likes(John,Peanuts)', 'Killed(Anil)']
--------------------------------------------
  Resolve ['¬Food(x)', 'Likes(John,x)'] and ['Food(y)', '¬Alive(x)', '¬Eats(x,y)']
  Substitution: {'x': 'y'}
  ⇒ ['¬Alive(y)', '¬Eats(y,y)', 'Likes(John,y)']
--------------------------------------------
  Resolve ['¬Eats(x,y)', 'Killed(x)', 'Food(y)'] and ['¬Food(Peanuts)']
  Substitution: {'y': 'Peanuts'}
```

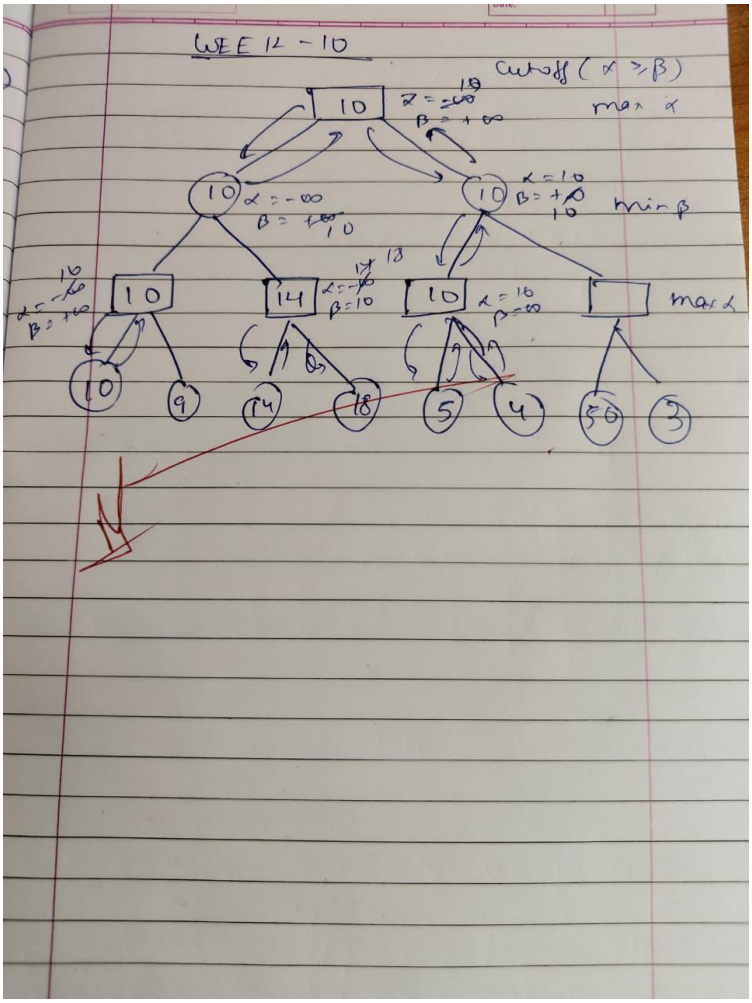
```
  Resolve ['¬Eats(x,y)', 'Killed(x)', 'Food(y)'] and ['¬Food(Peanuts)']
  Substitution: {'y': 'Peanuts'}
  ⇒ ['Killed(x)', '¬Eats(x,Peanuts)']
-----------------------------------------------
  Resolve ['¬Eats(x,y)', 'Killed(x)', 'Food(y)'] and ['¬Killed(Anil)']
  Substitution: {'x': 'Anil'}
  ⇒ ['Food(y)', '¬Eats(Anil,y)']
-----------------------------------------------
  Resolve ['¬Eats(x,y)', 'Killed(x)', 'Food(y)'] and ['Killed(x)', '¬Killed(x)']
  ⇒ ['Food(y)', 'Killed(x)', '¬Eats(x,y)']
-----------------------------------------------
  Resolve ['Eats(Anil,Peanuts)'] and ['Killed(y)', '¬Eats(y,y)', 'Likes(John,y)']
  Substitution: {'y': 'Peanuts'}
  ⇒ ['Likes(John,Peanuts)', 'Killed(Peanuts)']
-----------------------------------------------
  Resolve ['Eats(Anil,Peanuts)'] and ['Food(y)', '¬Alive(x)', '¬Eats(x,y)']
  Substitution: {'x': 'Anil', 'y': 'Peanuts'}
  ⇒ ['¬Alive(Anil)', 'Food(Peanuts)']
-----------------------------------------------
  Resolve ['¬Alive(x)', '¬Killed(x)'] and ['Killed(x)', '¬Killed(x)']
  ⇒ ['¬Killed(x)', '¬Alive(x)']
-----------------------------------------------
  Resolve ['Killed(x)', 'Alive(x)'] and ['Alive(x)', '¬Alive(x)']
  ⇒ ['Alive(x)', 'Killed(x)']
-----------------------------------------------
  Resolve ['¬Likes(John,Peanuts)'] and ['Killed(y)', '¬Eats(y,y)', 'Likes(John,y)']
  Substitution: {'y': 'Peanuts'}
  ⇒ ['Killed(Peanuts)', '¬Eats(Peanuts,Peanuts)']
-----------------------------------------------
  Resolve ['Killed(y)', '¬Eats(y,y)', 'Likes(John,y)'] and ['¬Killed(Anil)']
  Substitution: {'y': 'Anil'}
  ⇒ ['Likes(John,Anil)', '¬Eats(Anil,Anil)']
-----------------------------------------------
  Resolve ['¬Food(Peanuts)'] and ['Killed(Anil)', 'Food(Peanuts)']
  ⇒ ['Killed(Anil)']
-----------------------------------------------
  Resolve ['¬Food(Peanuts)'] and ['Food(y)', '¬Alive(x)', '¬Eats(x,y)']
  Substitution: {'y': 'Peanuts'}
  ⇒ ['¬Alive(x)', '¬Eats(x,Peanuts)']
```

```
    Resolve ['Killed(x)', 'Alive(x)'] and ['¬Alive(y)', '¬Eats(y,y)', 'Likes(John,y)']
    Substitution: {'y': 'x'}
    ⇒ ['Likes(John,x)', 'Killed(x)', '¬Eats(x,x)']
  -----------------------------------------------
    Resolve ['Killed(x)', 'Alive(x)'] and ['¬Killed(x)', '¬Alive(x)']
    ⇒ ['¬Killed(x)', 'Killed(x)']
  -----------------------------------------------
    Resolve ['¬Likes(John,Peanuts)'] and ['Likes(John,Peanuts)', 'Killed(Peanuts)']
    ⇒ ['Killed(Peanuts)']
  -----------------------------------------------
    Resolve ['¬Food(Peanuts)'] and ['Food(Peanuts)']
    ⇒ {} (empty clause)
  -----------------------------------------------

  ✅ Empty clause derived — query proven.

  ✅ Query Proven by Resolution: John likes peanuts.
```

Program 9

Implement Alpha-Beta Pruning.

Algorithm:



Code:

```
import math

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')
```

Output:

```
FINAL TREE

└── A (10)
    ├── 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 ()
```