VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Aadit Tomar (1BM23CS001)

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 **Aadit Tomar (1BM23CS001),** 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.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Infosys®
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Aadit Aadit

for successfully completing

Artificial Intelligence Foundation Certification

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

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

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

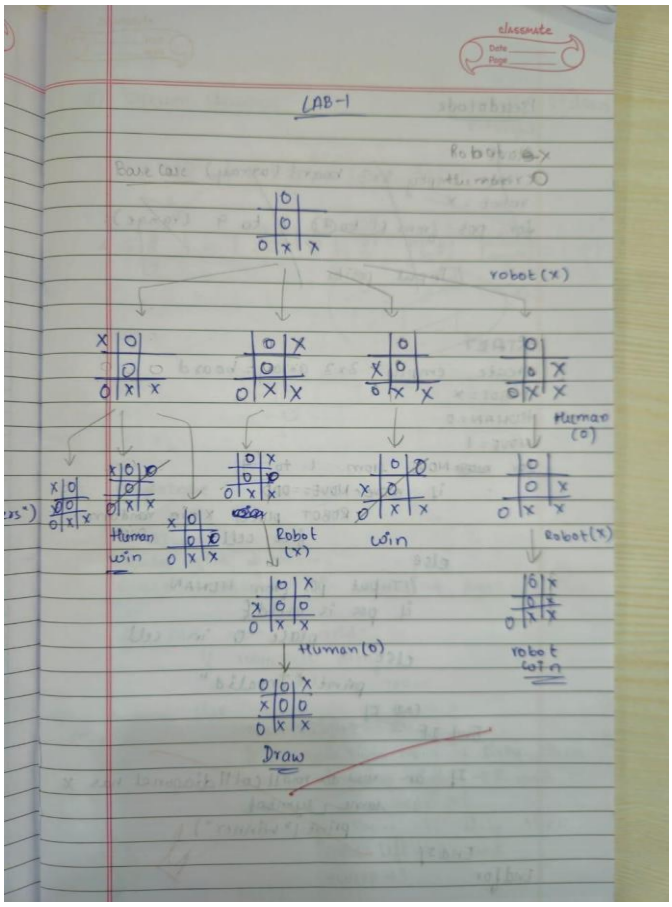
Github Link:

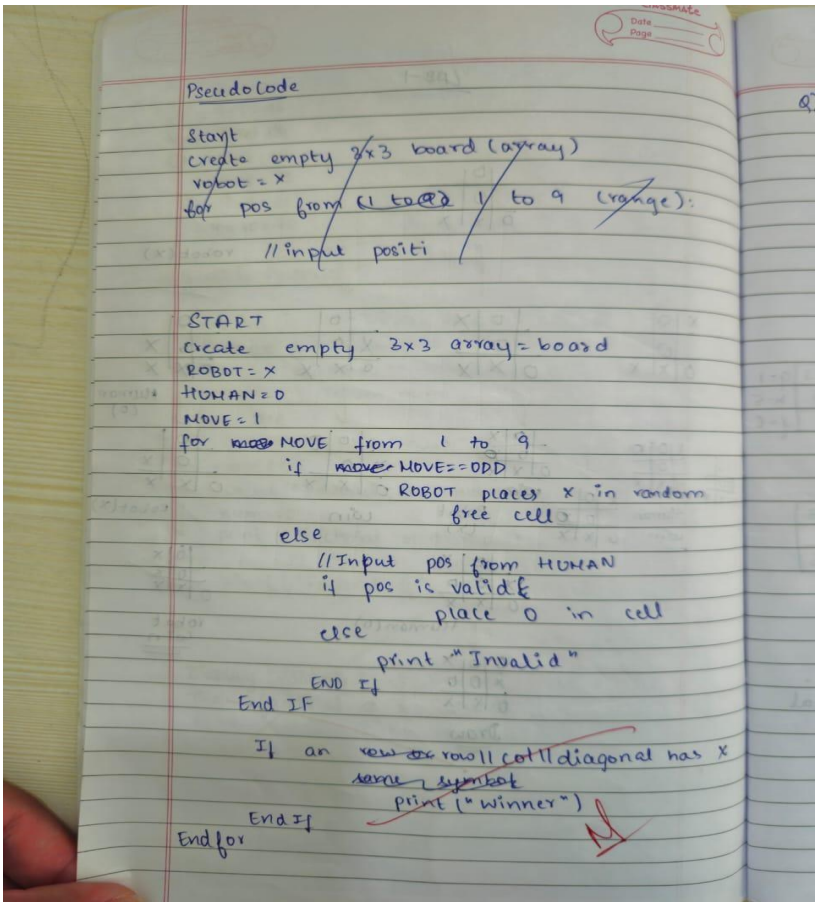
https://github.com/AADIT-DELL/AI_Lab

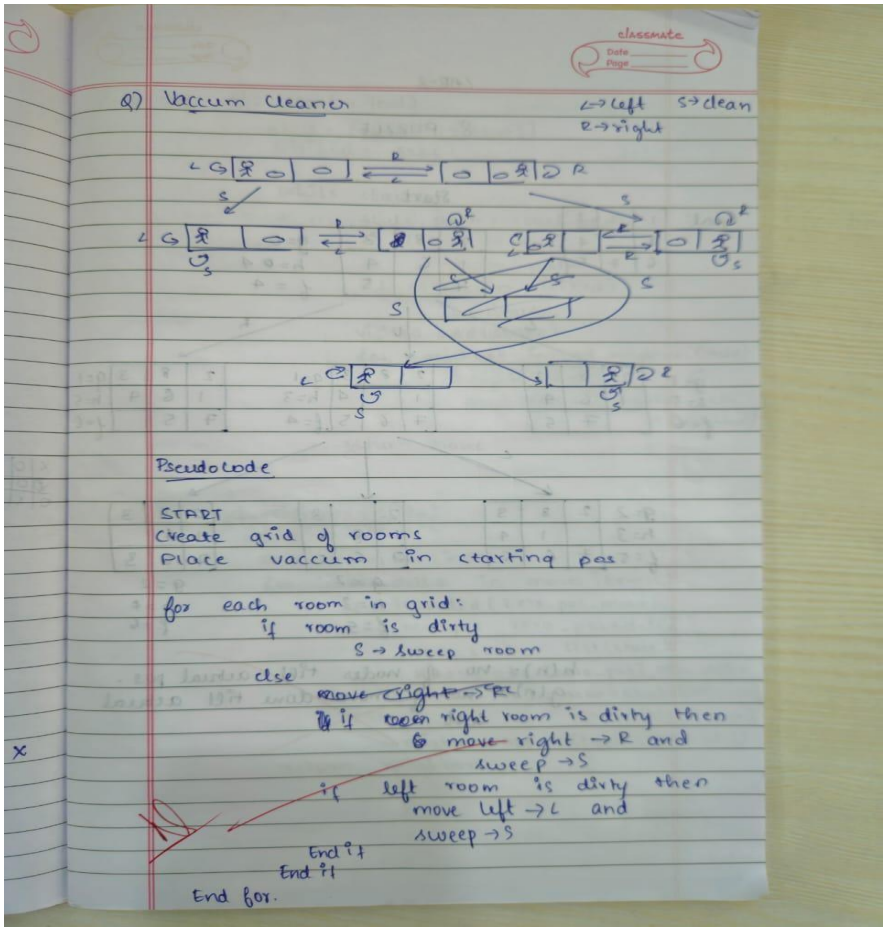
Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:







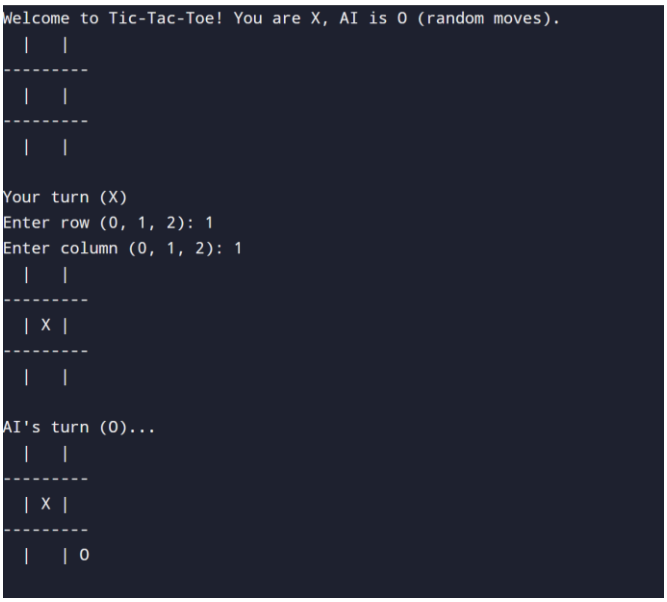
Code:

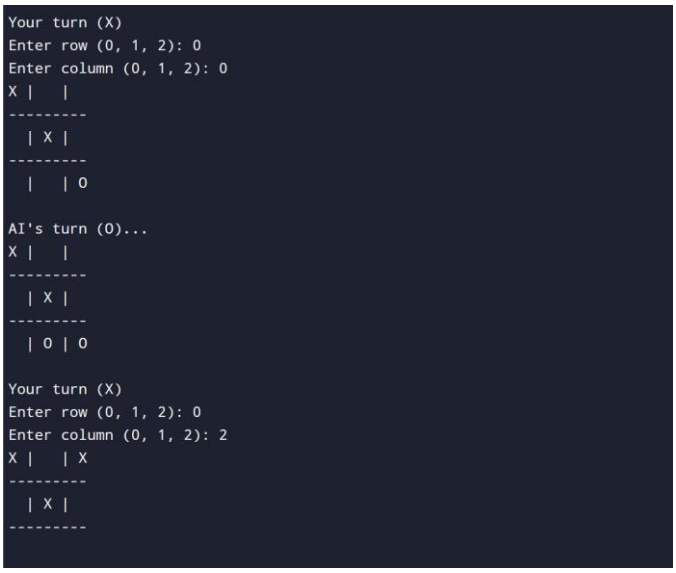
TIC-TAC-TOE

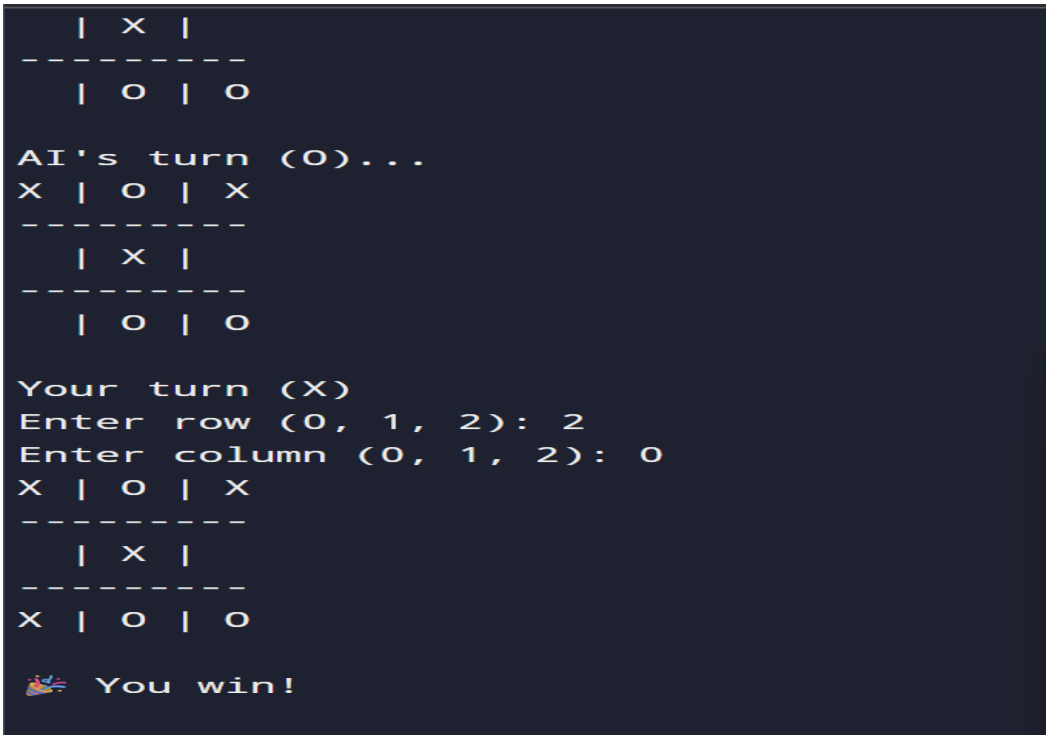
```
import random

def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    for r, row in enumerate(board):
        print(" | ".join(row))
        if r < 2:
            print("-" * 9)

def check_winner(board, player):
    """Checks if the player has won."""
    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):
    """Checks if the board is full."""
    return all(cell != " " for row in board for cell in row)

def ai_random_move(board):
    """AI chooses a random empty cell using numbers 0–8."""
    while True:
        pos = random.randint(0, 8)
        row, col = divmod(pos, 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 (random 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("\n🎉 You win!")
            break
        if is_full(board):
            print("\n🤝 It's a
            draw!") break

        print("\nAI's turn (O)...")
        move = ai_random_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()
```







VACCUM-CLEANER-

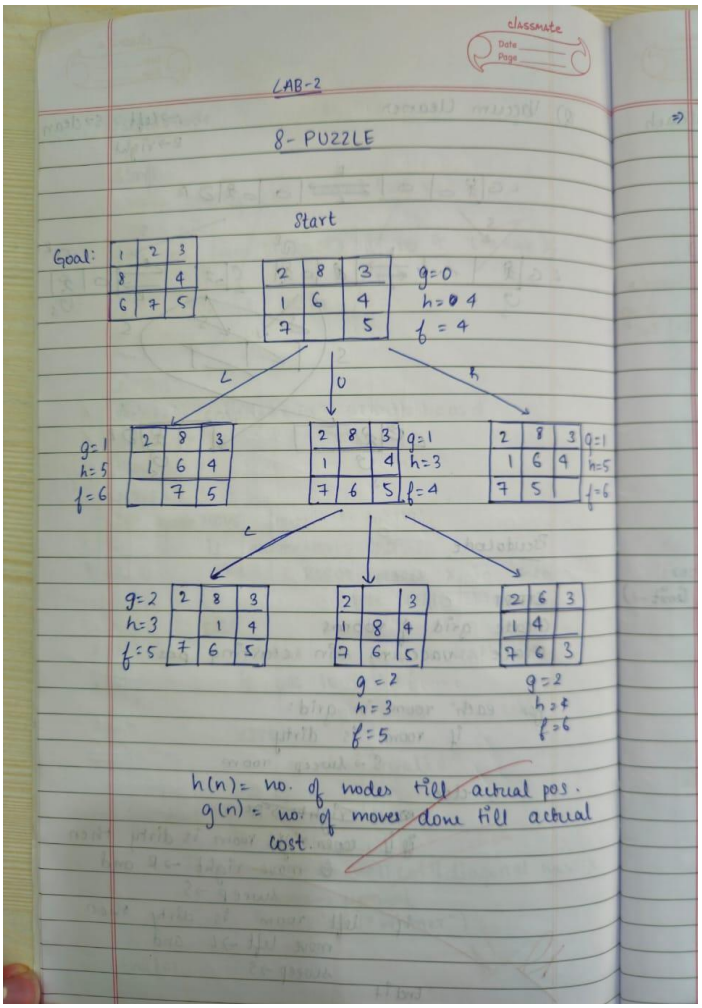
```
import random

environment = {
    "A": random.choice(["Clean", "Dirty"]),
    "B": random.choice(["Clean", "Dirty"])
}

def simple_reflex_agent(location, status):
    if status == "Dirty":
        return "Suck"
    elif location == "A":
        return "Right"
    else:
        return "Left"

def goal_based_agent(env):
    actions = []
    for location in ["A", "B"]:
        if env[location] == "Dirty":
            actions.append((location, "toClean"))
            env[location] = "Clean"
    return actions


def run_simulation():
    print("Initial Environment:", environment)

    location = random.choice(["A", "B"])
    action = simple_reflex_agent(location, environment[location])
    print(f"Reflex Agent at {location} sees {environment[location]} -> Action: {action}")

    actions = goal_based_agent(environment.copy())
    print("Goal-Based Agent Actions:", actions)

run_simulation()
```

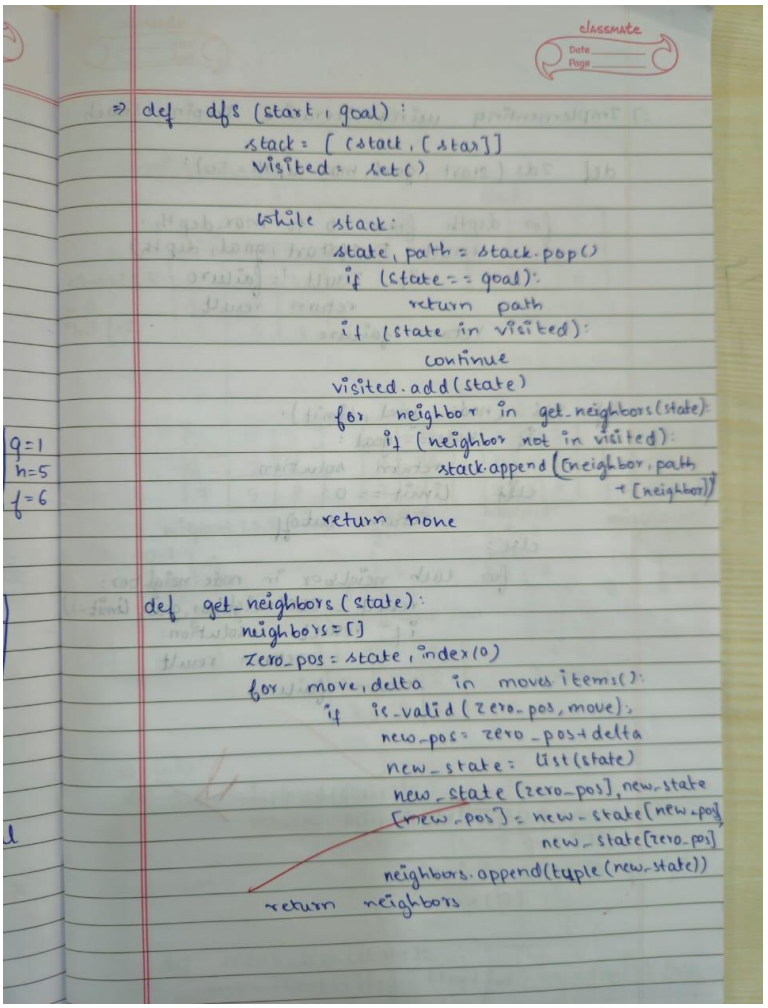
```
Initial Environment: {'A': 'Dirty', 'B': 'Clean'}
Reflex Agent at A sees Dirty -> Action: Suck
Goal-Based Agent Actions: [('A', 'toClean')]
```

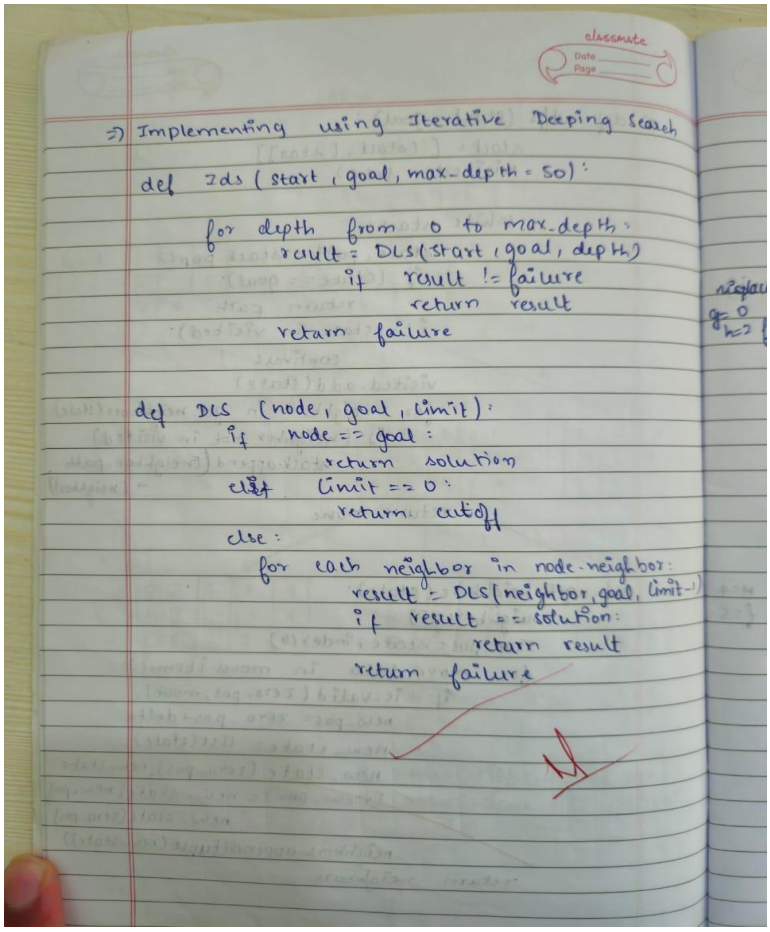
Program 2

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

ALGORITHM-





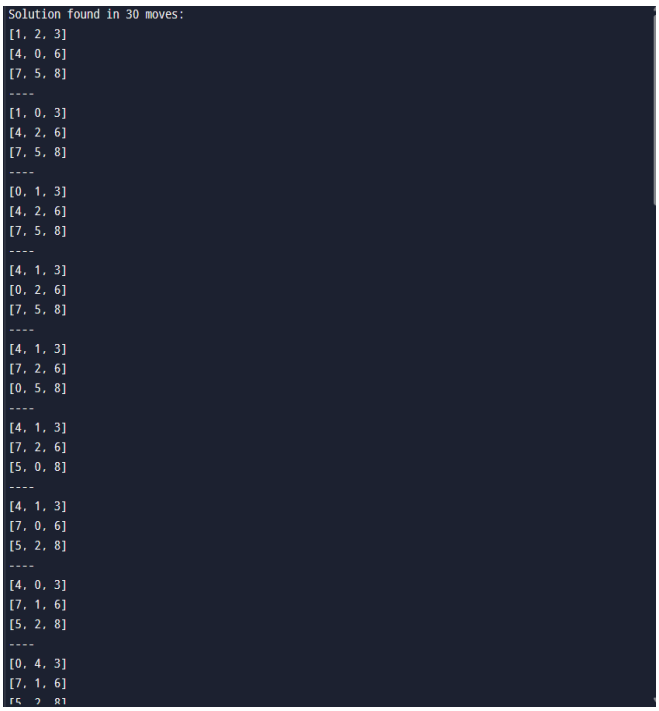


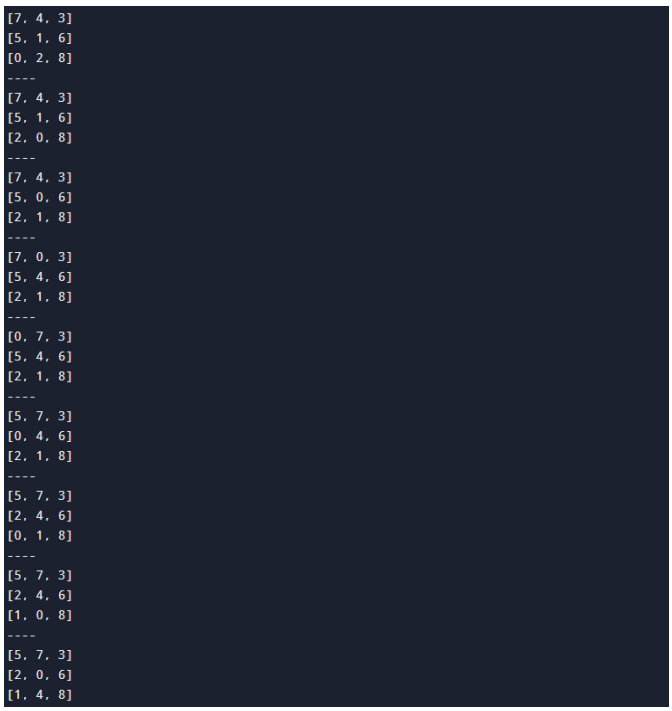
Code-

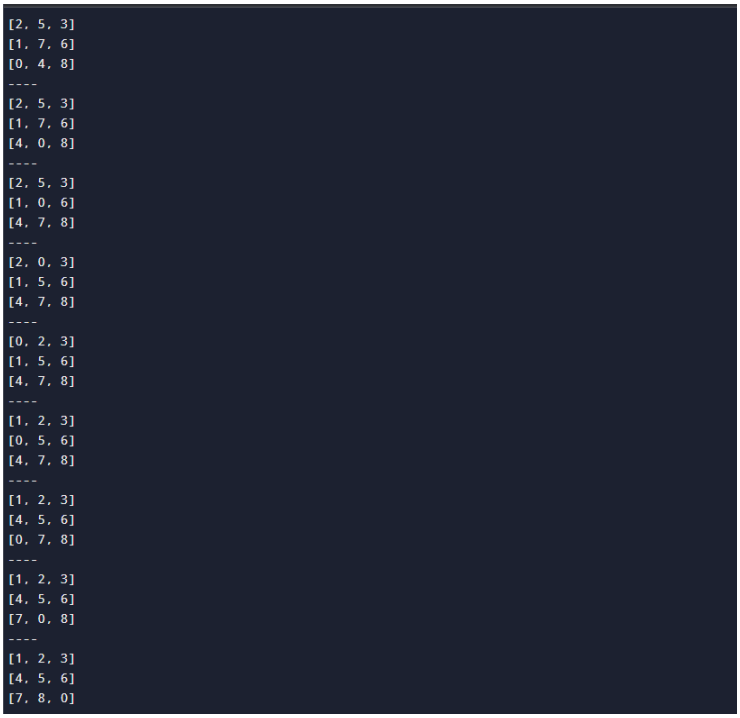
8-PUZZLE DFS-

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

moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

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

def get_neighbors(state):
    x, y = find_zero(state)
    neighbors = []
    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(state, visited):
    if state == goal:
        return [state]

    visited.add(str(state))

    for neighbor in get_neighbors(state):
        if str(neighbor) not in visited:
            path = dfs(neighbor, visited)
            if path:
                return [state] + path
    return None

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

solution = dfs(start, set())

if solution:
    print("Solution found in", len(solution) - 1, "moves:")
    for step in solution:
        for row in step:
            print(row)
        print("----")
else:
    print("No solution found")
```







8-PUZZLE IDS-

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

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

def neighbors(s):
    x, y = find_zero(s)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            ns = [r[:] for r in s]
            ns[x][y], ns[nx][ny] = ns[nx][ny], ns[x][y]
            yield ns

def dls(s, depth, visited):
    if s == goal:
        return [s]
    if depth == 0:
        return None
    visited.add(str(s))
    for n in neighbors(s):
```

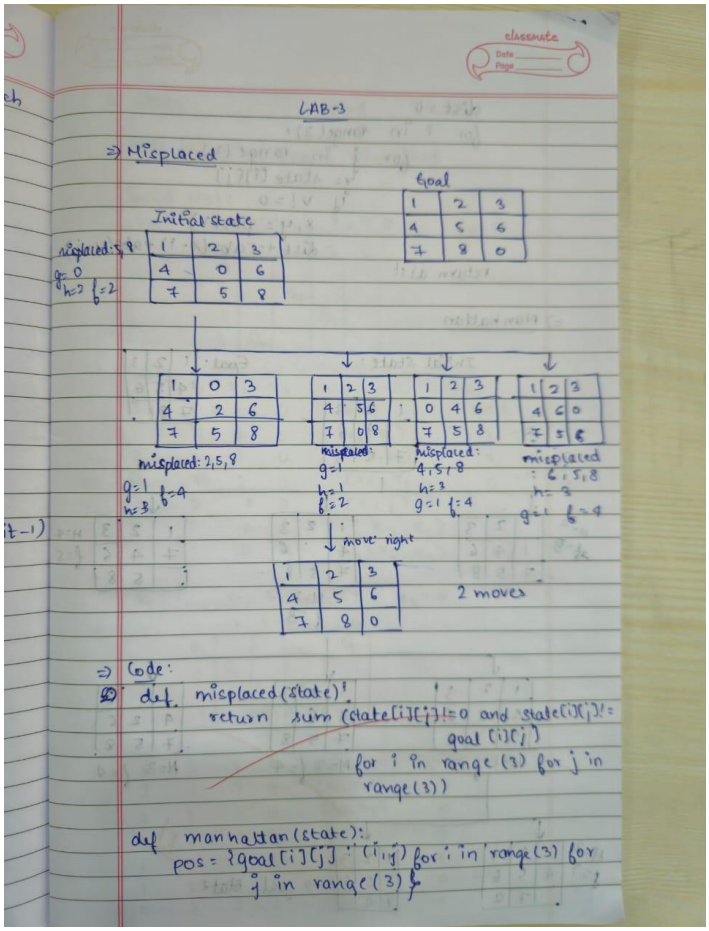
```
        if str(n) not in visited:
            path = dls(n, depth - 1, visited)
            if path:
                return [s] + path
    return None

def ids(start, limit=20):
    for d in range(limit + 1):
        path = dls(start, d, set())
        if path:
            return path
    return None

# Example
start = [[1,2,3],[4,0,6],[7,5,8]]
sol = ids(start)
if sol:
    print("Solved in", len(sol) - 1, "moves")
    for step in sol:
        for r in step:
            print(r)
        print(" --- ")
else:
    print("No solution")
```

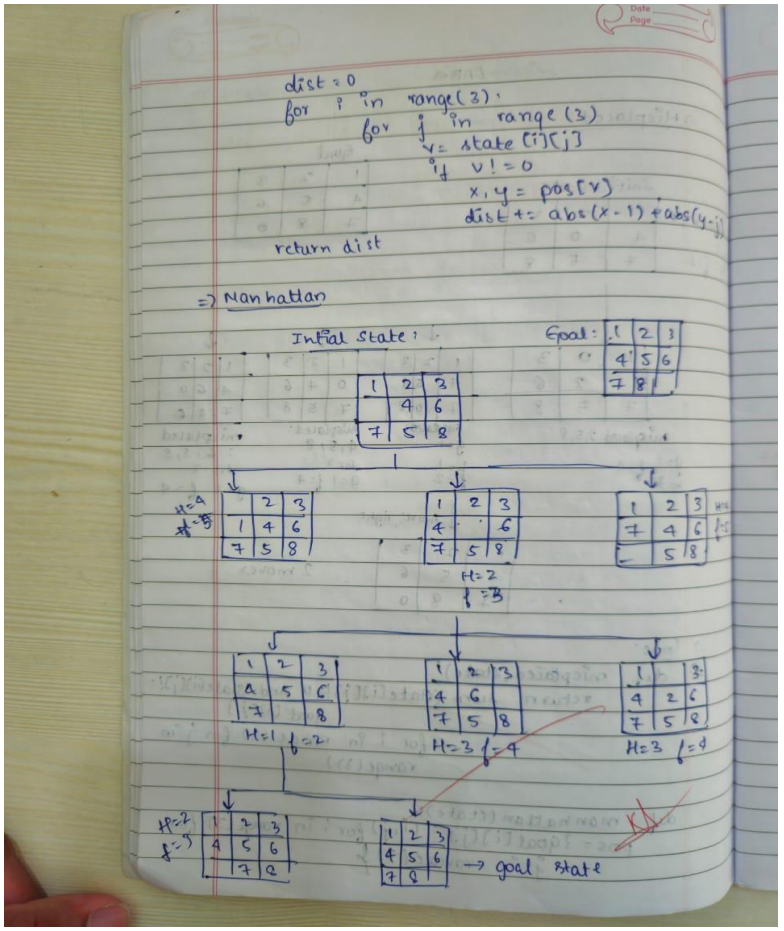
```
Solved in 2 moves
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
----
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
----
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
----

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

Program 3-

Implement A* search algorithm

Algorithm-





Code-

```
import heapq

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

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

def to_tuple(board):
    return tuple(tuple(row) for row in board)

def find_pos(board, value):
```

```
    for i in range(3):
        for j in range(3):
            if board[i][j] == value:
                return (i, j)

# Heuristic 1: misplaced tiles
def h_misplaced(board):
    count = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0 and board[i][j] != goal_state[i][j]:
                count += 1
    return count

# Heuristic 2: manhattan distance
def h_manhattan(board):
    dist = 0
    for i in range(3):
        for j in range(3):
            val = board[i][j]
            if val != 0:
                goal_i, goal_j = find_pos(goal_state, val)
                dist += abs(i - goal_i) + abs(j - goal_j)
    return dist

def get_neighbors(board):
    neighbors = []
    x, y = find_pos(board, 0)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_board = [list(row) for row in board]
            new_board[x][y], new_board[nx][ny] = new_board[nx][ny], new_board[x][y]
            neighbors.append(new_board)
    return neighbors

def print_board(board):
    for row in board:
        print(' '.join(str(x) for x in row))
    print()
```

```
def astar(start, heuristic):
    pq = []
    g = 0
    f = g + heuristic(start)
    heapq.heappush(pq, (f, g, start, []))
    visited = set()

    while pq:
        f, g, board, path = heapq.heappop(pq)
        if board == goal_state:
            return path + [board]

        visited.add(to_tuple(board))

        for neighbor in get_neighbors(board):
            if to_tuple(neighbor) not in visited:
                new_g = g + 1
                new_f = new_g + heuristic(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [board]))
    return None

start_state1 = [[1,2,3],
                [4,0,6],
                [7,5,8]]

start_state2 = [
    [2, 8, 3],
    [1, 6, 4],
    [7, 0, 5]
]

start_state3 = [
    [8, 0, 3],
    [2, 1, 4],
    [7, 6, 5]
]

print("Using Misplaced Tiles:")
solution = astar(start_state3, h_misplaced)
print("Steps:", len(solution)-1)
for step, board in enumerate(solution):
```

```
    print(f"Step {step}:")
    print_board(board)

print("Using Manhattan Distance:")
solution = astar(start_state3, h_manhattan)
print("Steps:", len(solution)-1)
for step, board in enumerate(solution):
    print(f"Step {step}:")
    print_board(board)
```

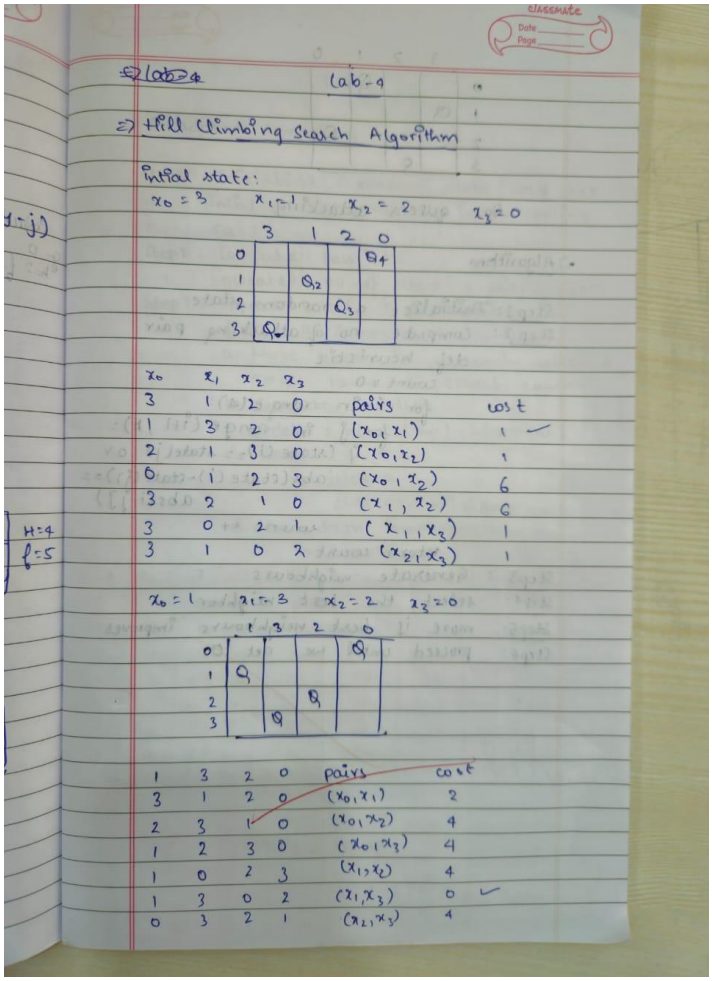
```
Using Misplaced Tiles:
Steps: 5
Step 0:
8 0 3
2 1 4
7 6 5

Step 1:
8 1 3
2 0 4
7 6 5

Step 2:
8 1 3
0 2 4
7 6 5

Step 3:
0 1 3
8 2 4
7 6 5

Step 4:
1 0 3
8 2 4
7 6 5

Step 5:
1 2 3
8 0 4
7 6 5
```

```
Using Manhattan Distance:
Steps: 5
Step 0:
8 0 3
2 1 4
7 6 5

Step 1:
8 1 3
2 0 4
7 6 5

Step 2:
8 1 3
0 2 4
7 6 5

Step 3:
0 1 3
8 2 4
7 6 5

Step 4:
1 0 3
8 2 4
7 6 5

Step 5:
1 2 3
8 0 4
7 6 5
```

Program 4-

Implement Hill Climbing search algorithm to solve N-Queens problem

ALGORITHM-



CODE-

```
import random

def display(board):
    n = len(board)
    for i in range(n):
        print(" ".join("Q" if board[i] == j else "." for j in range(n)))
    print()

def conflicts(board):
    n = len(board)
    count = 0
```

```
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                count += 1
    return count

def best_move(board):
    n = len(board)
    best = list(board)
    best_val = conflicts(board)
    for r in range(n):
        for c in range(n):
            if board[r] != c:
                temp = list(board)
                temp[r] = c
                val = conflicts(temp)
                if val < best_val:
                    best_val = val
                    best = temp
    return best, best_val

def hill_climb(n):
    state = [random.randint(0, n - 1) for _ in range(n)]
    cost = conflicts(state)
    print("Initial Board:")
    display(state)
    print(f"Initial Cost: {cost}\n")
    step = 1
    while True:
        nxt, nxt_cost = best_move(state)
        print(f"Step {step}:")
        print("Current Board:")
        display(state)
        print(f"Current Cost: {cost}")
        print(f"Best Neighbor Cost: {nxt_cost}\n")
        if nxt_cost >= cost:
            break
        state, cost = nxt, nxt_cost
        step += 1
    print("Final Board:")
    display(state)
```

```
    print(f"Final Cost: {cost}")
    if cost == 0:
        print("Goal State Reached!")
    else:
        print("Stuck in Local Minimum!")

hill_climb(4)
```

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

Initial Cost: 6

Step 1:
Current Board:
Q . . .
. Q . .
. . Q .
. . . Q

Current Cost: 6
Best Neighbor Cost: 4
```

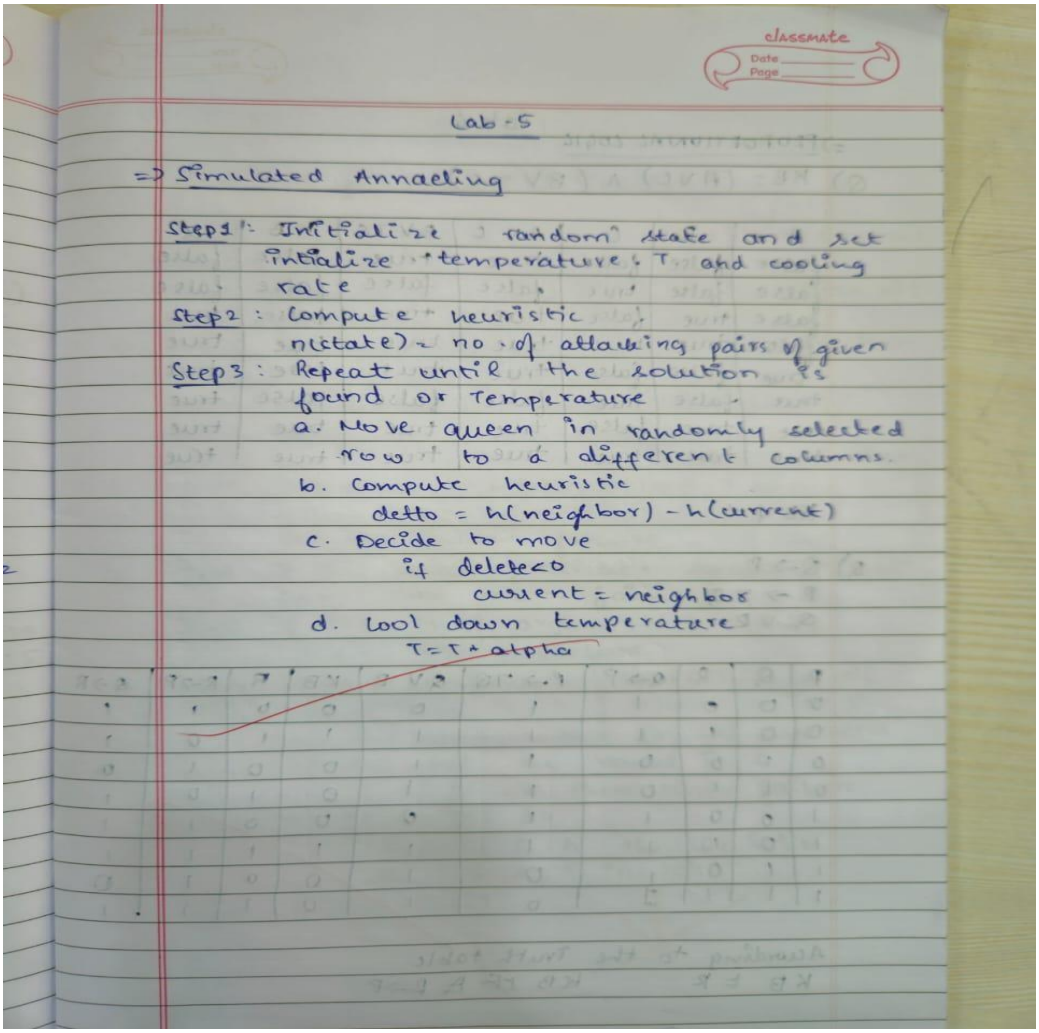
```
Step 2:
Current Board:
. Q . .
. Q . .
. . Q .
. . . Q

Current Cost: 4
Best Neighbor Cost: 2

Step 3:
Current Board:
. Q . .
Q . . .
. . Q .
. . . Q

Current Cost: 2
Best Neighbor Cost: 2
```

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

Final Cost: 2
Stuck in Local Minimum!

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

Program 5-

Simulated Annealing to Solve 8-Queens problem

Algorithm-



Lab - 5

=> Simulated Annealing

Step1 : Initialize random state and set initialize temperature : T and cooling rate

Step2 : Compute heuristic
h(state) = no. of attacking pairs of given

Step3 : Repeat until the solution is found or Temperature

a. Move queen in randomly selected row to a different columns.

b. Compute heuristic
delto = h(neighbor) - h(current)

c. Decide to move
if delete<0
current = neighbor

d. Cool down temperature
T = T * alpha

Code-

```
import random
import math

def show_board(queens):
    size = len(queens)
    for r in range(size):
        row = ["Q" if queens[r] == c else "." for c in range(size)]
        print(" ".join(row))
    print()

def conflicts(queens):
```

```
    size = len(queens)
    conflict_count = 0
    for r1 in range(size):
        for r2 in range(r1 + 1, size):
            same_col = queens[r1] == queens[r2]
            same_diag = abs(queens[r1] - queens[r2]) == abs(r1 - r2)
            if same_col or same_diag:
                conflict_count += 1
    return conflict_count

def get_random_neighbor(queens):
    size = len(queens)
    new_board = queens[:]
    row = random.randrange(size)
    col = random.randrange(size)
    new_board[row] = col
    return new_board

def anneal(n, start_temp=100, cooling=0.95, min_temp=1):
    current = [random.randrange(n) for _ in range(n)]
    energy = conflicts(current)
    temp = start_temp
    iteration = 1

    print("Initial Configuration:")
    show_board(current)
    print(f"Initial Conflicts: {energy}\n")

    while temp > min_temp and energy > 0:
        candidate = get_random_neighbor(current)
        next_energy = conflicts(candidate)
        delta = next_energy - energy

        if delta < 0 or random.random() < math.exp(-delta / temp):
            current = candidate
            energy = next_energy

        print(f"Iteration {iteration}: Temp={temp:.2f}, Conflicts={energy}")
        temp *= cooling
        iteration += 1

    print("\nFinal Configuration:")
    show_board(current)
    print(f"Final Conflicts: {energy}")

    if energy == 0:
        print("☑ Solution Found!")
```

```
    else:
        print("⚠ Stopped before finding a solution.")

anneal(8)
```

```
Initial Configuration:
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . Q .
. . . . . . . Q

Initial Conflicts: 5

Iteration 1: Temp=100.00, Conflicts=6
Iteration 2: Temp=95.00, Conflicts=9
Iteration 3: Temp=90.25, Conflicts=11
Iteration 4: Temp=85.74, Conflicts=9
Iteration 5: Temp=81.45, Conflicts=8
Iteration 6: Temp=77.38, Conflicts=6
Iteration 7: Temp=73.51, Conflicts=6
Iteration 8: Temp=69.83, Conflicts=4
Iteration 9: Temp=66.34, Conflicts=4
Iteration 10: Temp=63.02, Conflicts=5
Iteration 11: Temp=59.87, Conflicts=6
Iteration 12: Temp=56.88, Conflicts=7
Iteration 13: Temp=54.04, Conflicts=6
Iteration 14: Temp=51.33, Conflicts=8
Iteration 15: Temp=48.77, Conflicts=12
Iteration 16: Temp=46.33, Conflicts=9
Iteration 17: Temp=44.01, Conflicts=10
Iteration 18: Temp=41.81, Conflicts=9
Iteration 19: Temp=39.72, Conflicts=10
Iteration 20: Temp=37.74, Conflicts=9
Iteration 21: Temp=35.85, Conflicts=7
Iteration 22: Temp=34.06, Conflicts=8
Iteration 23: Temp=32.35, Conflicts=7
Iteration 24: Temp=30.74, Conflicts=10
Iteration 25: Temp=29.20, Conflicts=11
Iteration 26: Temp=27.74, Conflicts=11
Iteration 27: Temp=26.35, Conflicts=12
Iteration 28: Temp=25.03, Conflicts=13
Iteration 29: Temp=23.78, Conflicts=12
Iteration 30: Temp=22.59, Conflicts=9
Iteration 31: Temp=21.46, Conflicts=7
```

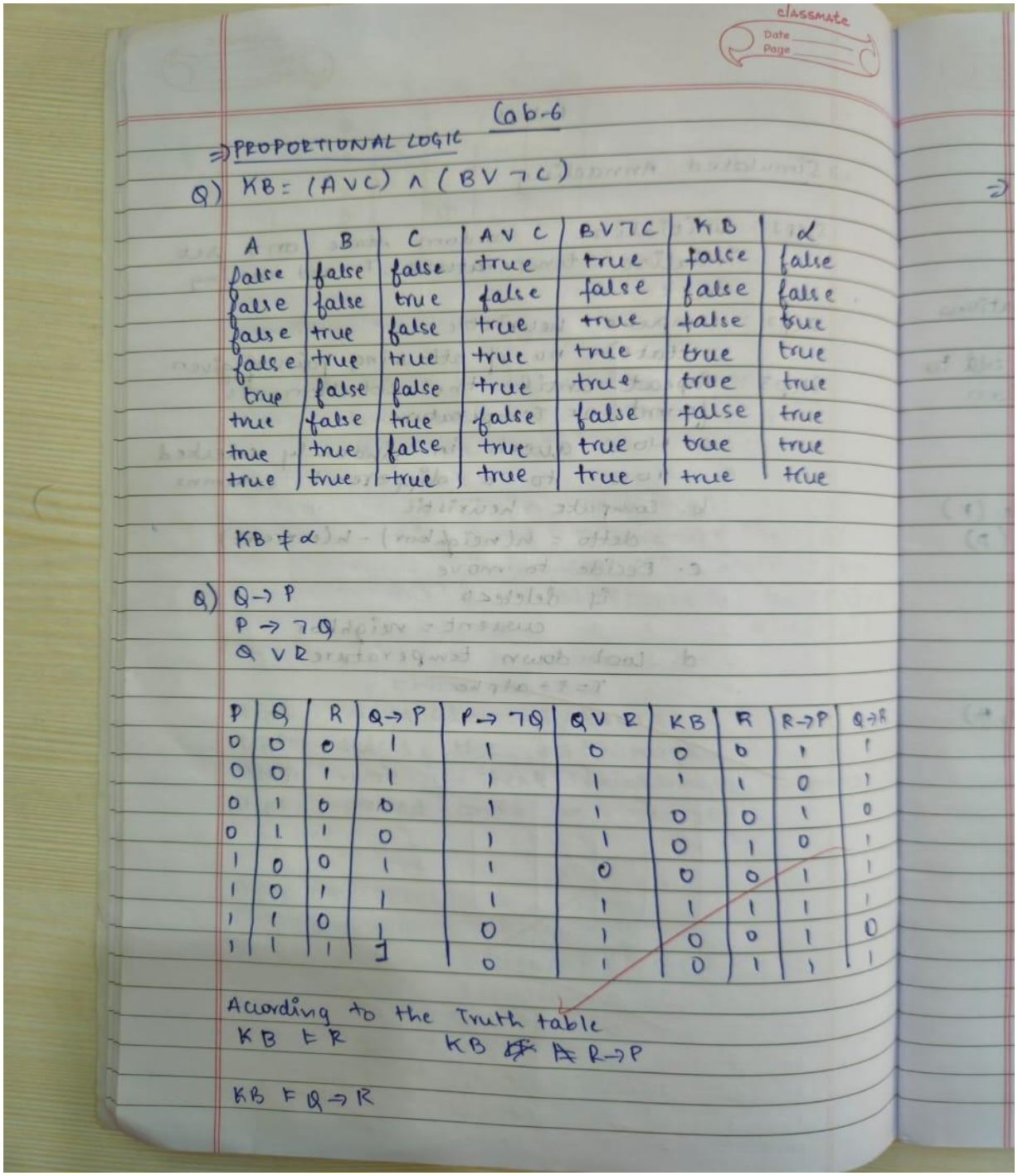
```
Iteration 30: Temp=22.59, Conflicts=9
Iteration 31: Temp=21.46, Conflicts=7
Iteration 32: Temp=20.39, Conflicts=6
Iteration 33: Temp=19.37, Conflicts=8
Iteration 34: Temp=18.40, Conflicts=8
Iteration 35: Temp=17.48, Conflicts=5
Iteration 36: Temp=16.61, Conflicts=7
Iteration 37: Temp=15.78, Conflicts=9
Iteration 38: Temp=14.99, Conflicts=9
Iteration 39: Temp=14.24, Conflicts=8
Iteration 40: Temp=13.53, Conflicts=7
Iteration 41: Temp=12.85, Conflicts=8
Iteration 42: Temp=12.21, Conflicts=10
Iteration 43: Temp=11.60, Conflicts=8
Iteration 44: Temp=11.02, Conflicts=10
Iteration 45: Temp=10.47, Conflicts=10
Iteration 46: Temp=9.94, Conflicts=10
Iteration 47: Temp=9.45, Conflicts=9
Iteration 48: Temp=8.97, Conflicts=8
Iteration 49: Temp=8.53, Conflicts=8
Iteration 50: Temp=8.10, Conflicts=8
Iteration 51: Temp=7.69, Conflicts=11
Iteration 52: Temp=7.31, Conflicts=11
Iteration 53: Temp=6.94, Conflicts=11
Iteration 54: Temp=6.60, Conflicts=9
Iteration 55: Temp=6.27, Conflicts=9
Iteration 56: Temp=5.95, Conflicts=9
Iteration 57: Temp=5.66, Conflicts=9
Iteration 58: Temp=5.37, Conflicts=9
Iteration 59: Temp=5.10, Conflicts=8
Iteration 60: Temp=4.85, Conflicts=6
Iteration 61: Temp=4.61, Conflicts=7
Iteration 62: Temp=4.38, Conflicts=6
Iteration 63: Temp=4.16, Conflicts=6
Iteration 64: Temp=3.95, Conflicts=6
Iteration 65: Temp=3.75, Conflicts=6
Iteration 66: Temp=3.56, Conflicts=8
Iteration 67: Temp=3.39, Conflicts=8
Iteration 68: Temp=3.22, Conflicts=6
Iteration 69: Temp=3.06, Conflicts=6
Iteration 70: Temp=2.90, Conflicts=6
Iteration 71: Temp=2.76, Conflicts=6
Iteration 72: Temp=2.62, Conflicts=6
```

```
Iteration 63: Temp=4.16, Conflicts=6
Iteration 64: Temp=3.95, Conflicts=6
Iteration 65: Temp=3.75, Conflicts=6
Iteration 66: Temp=3.56, Conflicts=8
Iteration 67: Temp=3.39, Conflicts=8
Iteration 68: Temp=3.22, Conflicts=6
Iteration 69: Temp=3.06, Conflicts=6
Iteration 70: Temp=2.90, Conflicts=6
Iteration 71: Temp=2.76, Conflicts=6
Iteration 72: Temp=2.62, Conflicts=6
Iteration 73: Temp=2.49, Conflicts=6
Iteration 74: Temp=2.36, Conflicts=7
Iteration 75: Temp=2.25, Conflicts=7
Iteration 76: Temp=2.13, Conflicts=7
Iteration 77: Temp=2.03, Conflicts=7
Iteration 78: Temp=1.93, Conflicts=7
Iteration 79: Temp=1.83, Conflicts=7
Iteration 80: Temp=1.74, Conflicts=7
Iteration 81: Temp=1.65, Conflicts=7
Iteration 82: Temp=1.57, Conflicts=7
Iteration 83: Temp=1.49, Conflicts=6
Iteration 84: Temp=1.42, Conflicts=4
Iteration 85: Temp=1.35, Conflicts=4
Iteration 86: Temp=1.28, Conflicts=4
Iteration 87: Temp=1.21, Conflicts=4
Iteration 88: Temp=1.15, Conflicts=4
Iteration 89: Temp=1.10, Conflicts=4
Iteration 90: Temp=1.04, Conflicts=4

Final Configuration:
. . . . . Q
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
. . . . . Q . .
. . . . . Q .
. Q . . . . . .
. . . . . Q .

Final Conflicts: 4
⚠Stopped before finding a solution.
```

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
import re

def interpret(formula, assignment):
    s = formula.replace("<->", " == ")
    s = s.replace("->", " <= ")
    s = re.sub(r'~(\w+)', r'(not \1)', s)
    s = re.sub(r'~\(([^)]+)\)', r'(not (\1))', s)
    s = s.replace("^", " and ")
    s = s.replace("v", " or ")
    for sym, val in assignment.items():
        s = re.sub(r'\b' + re.escape(sym) + r'\b', str(val), s)
    return eval(s)

def tt_entails(kb, query, symbols):
    ok = True
    assignments = list(itertools.product([True, False], repeat=len(symbols)))
    print("Truth Table Evaluation:\n")
    header = " | ".join(symbols) + " | KB | Query | KB ⇒ Query"
    print(header)
    print("-" * (len(header) * 2))
    for values in assignments:
        model = dict(zip(symbols, values))
        kb_val = interpret(kb, model)
        q_val = interpret(query, model)
        implies = (not kb_val) or q_val
        if kb_val and not q_val:
            ok = False
        row = " | ".join('T' if v else 'F' for v in values)
        row += f" | {'T' if kb_val else 'F'}  | {'T' if q_val else 'F'}   | {'T' if implies else
'F'}"
        print(row)
    print("\nResult:")
    if ok:
        print("The Knowledge Base entails the Query (KB ⊨ Query)")
    else:
        print("The Knowledge Base does NOT entail the Query (KB ⊭ Query)")

kb = "(Q -> P) ^ (P -> ~Q) ^ (Q v R)"
syms = ["P", "Q", "R"]
qs = ["R", "R -> P", "Q -> R"]

for q in qs:
    print(f"\nEvaluating Query: {q}\n")
    tt_entails(kb, q, syms)
```

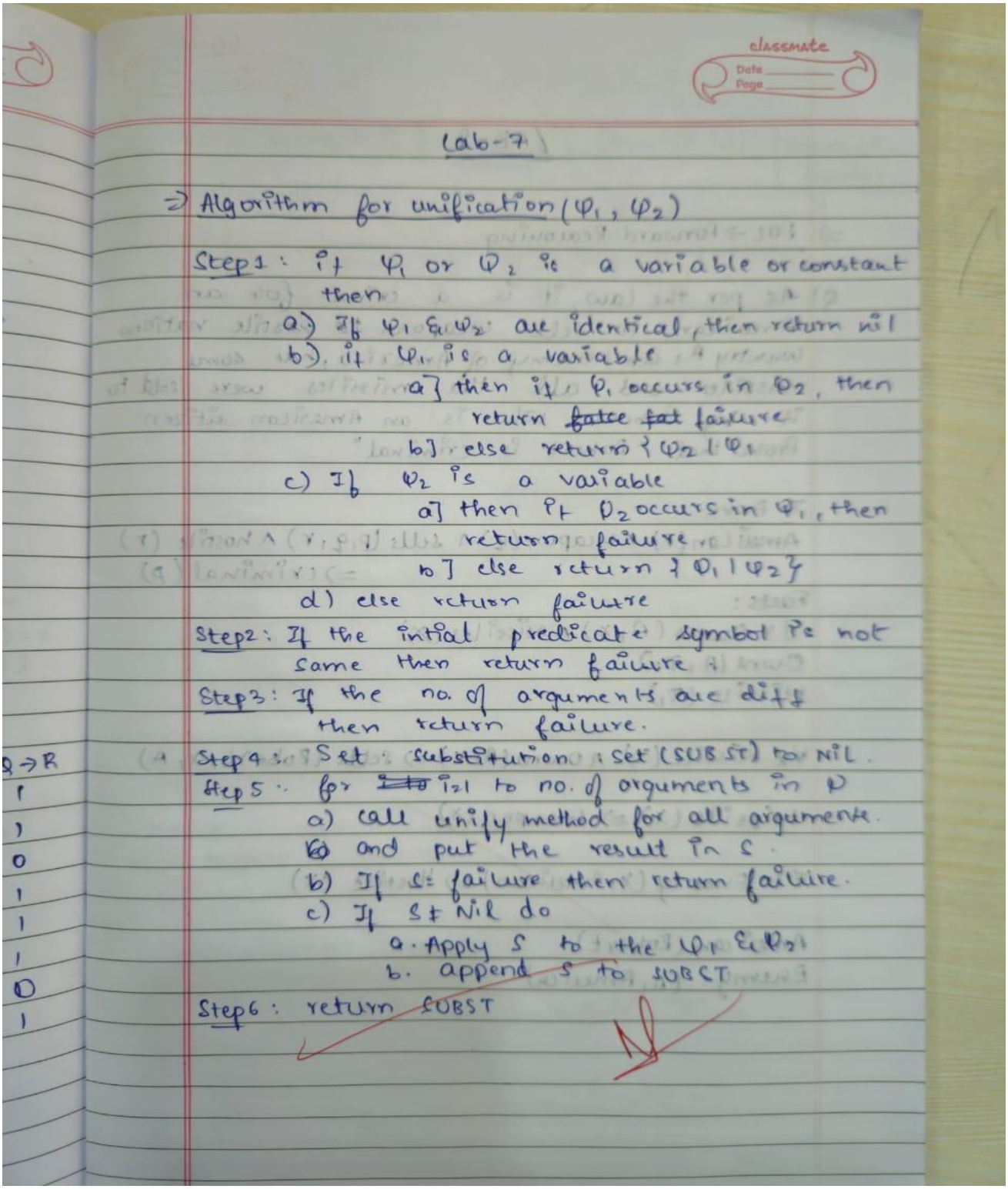
```
print("\n" + "=" * 50 + "\n")
```

```
Evaluating Query: R

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
------------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | F     | T
T | F | T | T  | T     | T
T | F | F | F  | F     | T
F | T | T | F  | T     | T
F | T | F | F  | F     | T
F | F | T | T  | T     | T
F | F | F | F  | F     | T

Result:
The Knowledge Base entails the Query (KB ⊨ Query)

==================================================


Evaluating Query: R -> P

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
------------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | T     | T
T | F | T | T  | T     | T
T | F | F | F  | T     | T
F | T | T | F  | F     | T
F | T | F | F  | T     | T
F | F | T | T  | F     | F
F | F | F | F  | T     | T

Result:
The Knowledge Base does NOT entail the Query (KB ⊭ Query)

==================================================
```

Program 7-

Implement unification in first order logic

ALGORITHM-



Lab-7

⇒ Algorithm for unification (ψ_1, ψ_2)

Step 1 : if ψ_1 or ψ_2 is a variable or constant then

a) If ψ_1 & ψ_2 are identical, then return nil

b) if ψ_1 is a variable

a] then if ψ_1 occurs in ψ_2, then return ~~false~~ ~~fat~~ failure

b] else return { ψ_2 / ψ_1 }

c) If ψ_2 is a variable

a] then if ψ_2 occurs in ψ_1, then return failure

b] else return { ψ_1 / ψ_2 }

d) else return failure

Step 2 : If the intial predicate symbol is not same then return failure

Step 3 : If the no. of arguments are diff then return failure.

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

Step 5 : for ~~i=0~~ i=1 to no. of arguments in ψ

a) call unify method for all arguments.

b) and put the result in S.

b) If S= failure then return failure.

c) If S ≠ Nil do

a. Apply S to the ψ_1 & ψ_2

b. append S to SUBST

Step 6 : return SUBST

Code-

```
def is_var(x):
    return isinstance(x, str) and x.islower()

def is_const(x):
    return isinstance(x, str) and x[:1].isupper()

def occurs_in(v, expr, subst):
    if v == expr:
        return True
    if isinstance(expr, list):
        return any(occurs_in(v, part, subst) for part in expr)
    if expr in subst:
        return occurs_in(v, subst[expr], subst)
    return False

def mgu(a, b, subst=None, depth=0):
    pad = "  " * depth
    if subst is None:
        print(pad + "No substitution available. Fail.")
        return None

    print(pad + f"Unify {a} with {b} | subst={subst}")

    if a == b:
        print(pad + "Same terms. Keep subst.")
        return subst

    if is_var(a):
        return bind_var(a, b, subst, depth)

    if is_var(b):
        return bind_var(b, a, subst, depth)

    if isinstance(a, list) and isinstance(b, list):
        if len(a) != len(b):
            print(pad + "Arity mismatch. Fail.")
            return None
        for u, v in zip(a, b):
            subst = mgu(u, v, subst, depth + 1)
            if subst is None:
                print(pad + "Element unification failed.")
                return None
        return subst

    print(pad + "Incompatible atoms/structures. Fail.")
    return None
```

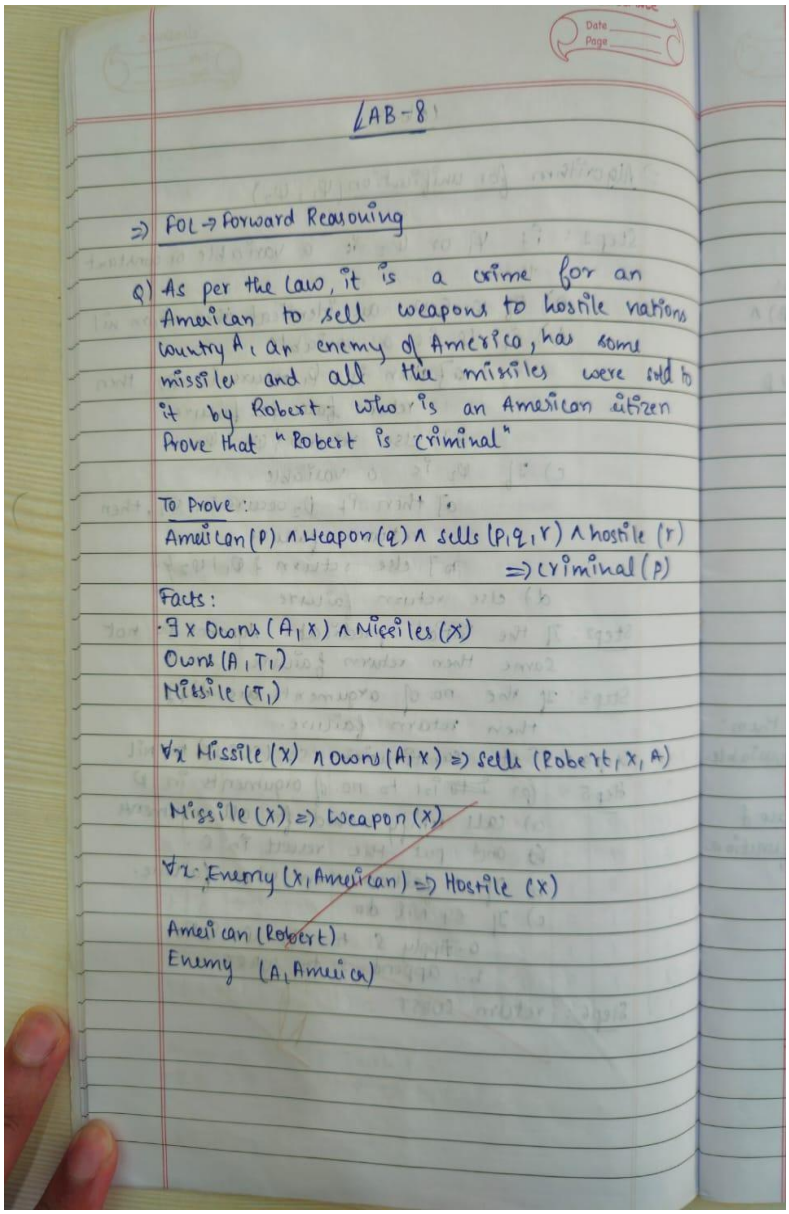
```
def bind_var(v, x, subst, depth):
    pad = "  " * depth
    if v in subst:
        print(pad + f"{v} already mapped; unify {subst[v]} with {x}")
        return mgu(subst[v], x, subst, depth + 1)
    if is_var(x) and x in subst:
        print(pad + f"{x} already mapped; unify {v} with {subst[x]}")
        return mgu(v, subst[x], subst, depth + 1)
    if occurs_in(v, x, subst):
        print(pad + f"Occurs-check: {v} in {x}. Fail.")
        return None
    print(pad + f"Extend: {v} -> {x}")
    subst[v] = x
    return subst

expr1 = ['f', 'X', ['g', 'Y']]
expr2 = ['f', 'a', ['g', 'b']]

print("Starting Unification:\n")
result = mgu(expr1, expr2, subst={})
print("\nFinal Unification Result:", result)
```

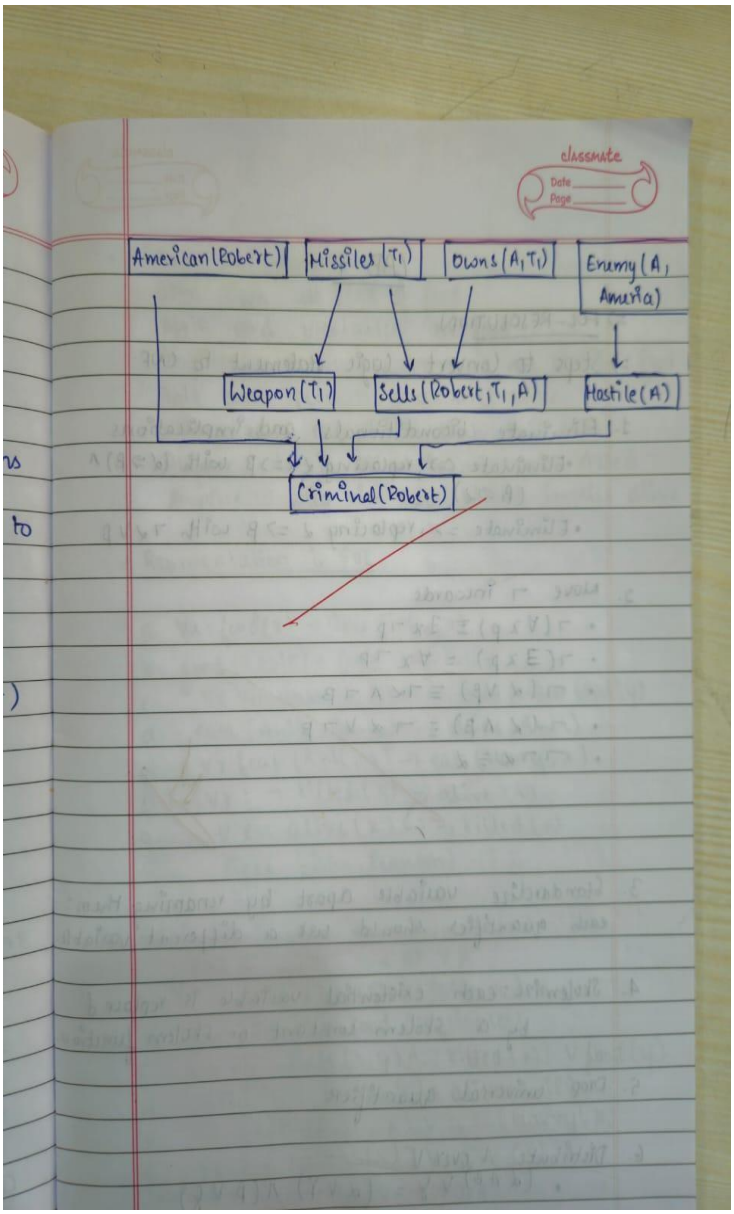
```
Starting Unification:

Unify ['f', 'X', ['g', 'Y']] with ['f', 'a', ['g', 'b']] | subst={}
  Unify f with f | subst={}
  Same terms. Keep subst.
  Unify X with a | subst={}
  Extend: a -> X
  Unify ['g', 'Y'] with ['g', 'b'] | subst={'a': 'X'}
    Unify g with g | subst={'a': 'X'}
    Same terms. Keep subst.
    Unify Y with b | subst={'a': 'X'}
    Extend: b -> Y

Final Unification Result: {'a': 'X', 'b': 'Y'}
```

Program 8-

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

ALGORITHM-





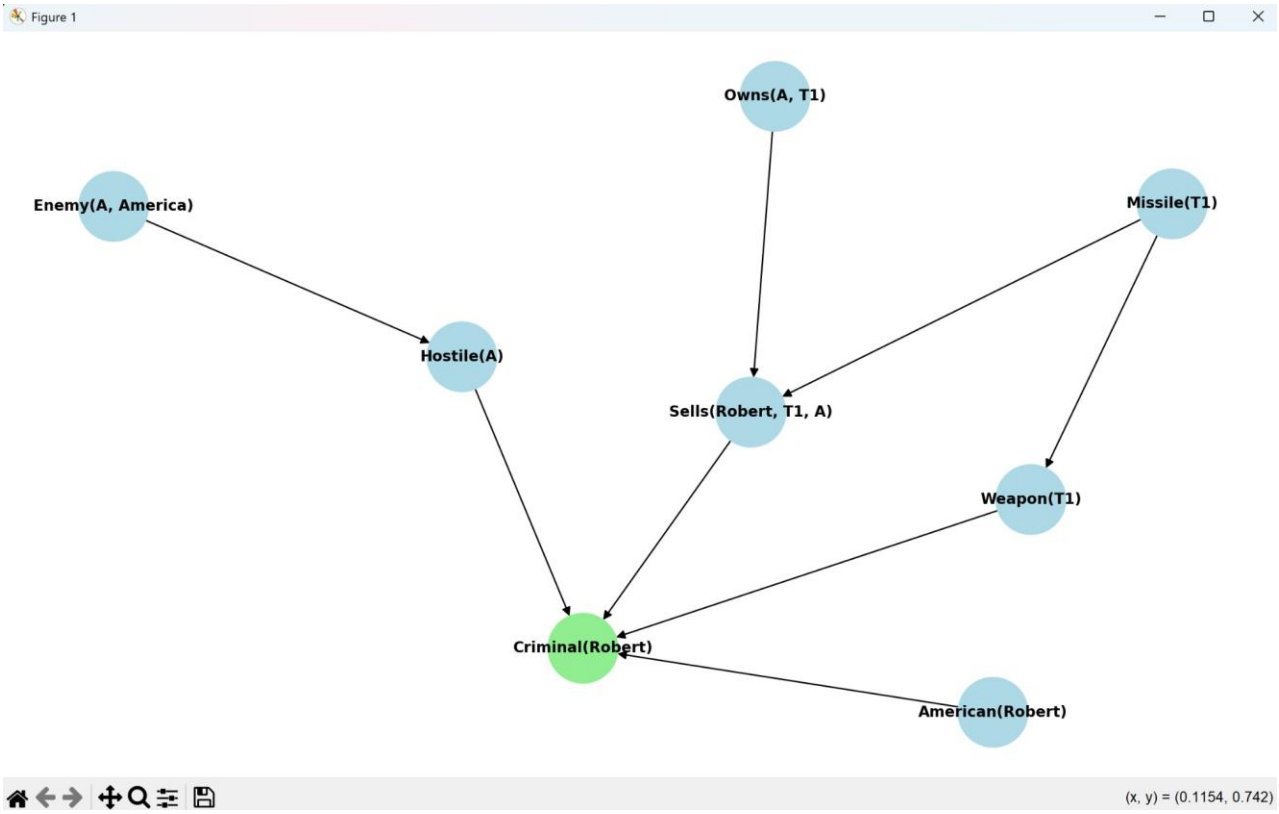
Code-

```
# -*- coding: utf-8 -*-
from copy import deepcopy
import networkx as nx
import matplotlib.pyplot as plt
```

```
KB = [
   {"if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"], "then": "Criminal(p)"},
   {"if": ["Missile(x)"], "then": "Weapon(x)"},
   {"if": ["Enemy(x, America)"], "then": "Hostile(x)"},
   {"if": ["Missile(x)", "Owns(A, x)"], "then": "Sells(Robert, x, A)"},
   {"fact": "American(Robert)"},
   {"fact": "Enemy(A, America)"},
   {"fact": "Missile(T1)"},
   {"fact": "Owns(A, T1)"}
]

goal = "Criminal(Robert)"

def split_pred(s):
   name, rest = s.split("(", 1)
   args = rest[:-1].split(",")
   return name.strip(), [a.strip() for a in args]

def sym_is_var(t):
   return t[0].islower()

def match_terms(a, b, theta=None):
   if theta is None:
      theta = {}
   p1, args1 = split_pred(a)
   p2, args2 = split_pred(b)
   if p1 != p2 or len(args1) != len(args2):
      return None
   for u, v in zip(args1, args2):
      if u == v:
         continue
      if sym_is_var(u):
         theta[u] = v
      elif sym_is_var(v):
         theta[v] = u
      else:
         return None
   return theta

def apply(subst, atom):
   name, args = split_pred(atom)
   out = []
   for a in args:
      while a in subst:
         a = subst[a]
      out.append(a)
```

```
    return f"{name}({', '.join(out)})"

def fc_prove(kb, q):
    facts = {e["fact"] for e in kb if "fact" in e}
    rules = [e for e in kb if "if" in e]
    edges = []

    print("Known facts:")
    for f in facts:
        print("  ", f)
    print()

    changed = True
    while changed:
        changed = False
        for rule in rules:
            prem = rule["if"]
            concl = rule["then"]
            sigma_list = [{}]
            for lit in prem:
                next_sigma = []
                for sig in sigma_list:
                    lit_inst = apply(sig, lit)
                    for f in facts:
                        m = match_terms(lit_inst, f, deepcopy(sig))
                        if m is not None:
                            next_sigma.append(m)
                sigma_list = next_sigma
                if not sigma_list:
                    break
            for sig in sigma_list:
                new_fact = apply(sig, concl)
                if new_fact not in facts:
                    print(f"Derived: {new_fact}")
                    facts.add(new_fact)
                    changed = True
                    for lit in prem:
                        edges.append((apply(sig, lit), new_fact))
                    if match_terms(new_fact, q):
                        print("\nGoal satisfied:", q)
                        draw_inference(edges, q)
                        return True

    print("\nGoal not provable.")
    draw_inference(edges, q)
    return False
```

```
def draw_inference(edge_list, target):
    G = nx.DiGraph()
    G.add_edges_from(edge_list)
    plt.figure(figsize=(12, 7))
    pos = nx.spring_layout(G, seed=42)
    colors = ["lightgreen" if n == target else "lightblue" for n in G.nodes()]
    nx.draw(G, pos, with_labels=True, node_color=colors, node_size=2200,
            font_size=10, font_weight="bold", arrows=True, arrowstyle="-|>", arrowsize=12)
    plt.title("Forward Chaining Inference Graph", fontsize=14, fontweight="bold")
    plt.show()

print("\n--- Forward Chaining (FOL-FC-ASK) ---\n")
fc_prove(KB, goal)
```

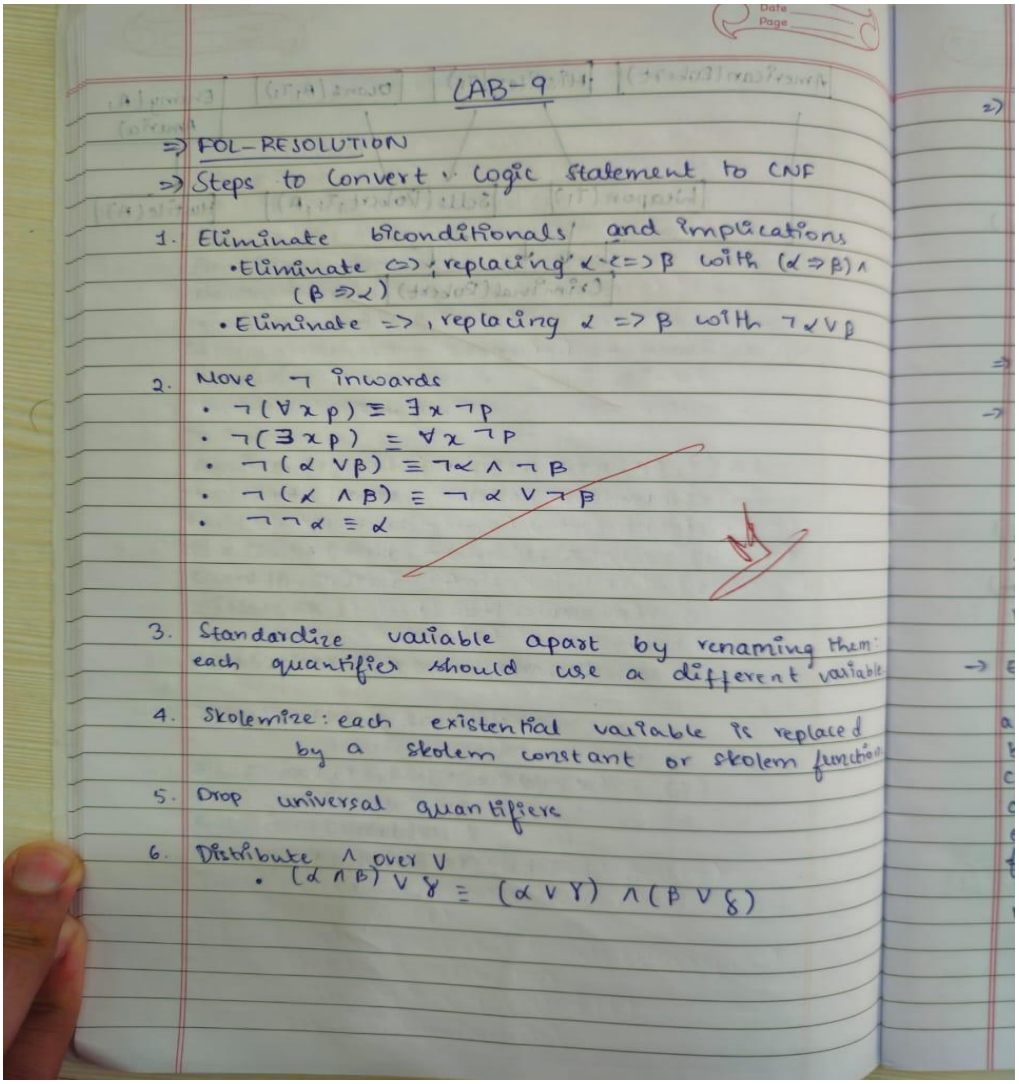


```
--- Forward Chaining (FOL-FC-ASK) ---

Known facts:
  Missile(T1)
  Owns(A, T1)
  Enemy(A, America)
  American(Robert)

Derived: Weapon(T1)
Derived: Hostile(A)
Derived: Sells(Robert, T1, A)
Derived: Criminal(Robert)

Goal not provable.
```

Program 9-

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

ALGORITHM-



LAB-9

=> FOL-RESOLUTION

=> Steps to Convert Logic Statement to CNF

1. Eliminate biconditionals 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 $\neg$ inwards
   - $\neg(\forall x\, p) \equiv \exists x \neg p$
   - $\neg(\exists x\, p) \equiv \forall x \neg p$
   - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
   - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
   - $\neg\neg\alpha \equiv \alpha$
3. Standardize variable apart by renaming them: each quantifier should use a different variable
4. Skolemize: each existential variable is replaced by a skolem constant or skolem function
5. Drop universal quantifiers
6. Distribute $\wedge$ over $\vee$
   - $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

⇒ Given KB/ premises:

John likes all kind of food
Apple and vegetables are food
Anything anyone eats and not killed is food
Anil eats peanuts and still alive
Harry eats everything that Anil eats
Anyone who is alive implies not killed
Anyone who is not killed implies alive

⇒ Representation in FOL

→ a. $\forall x : food(x) \rightarrow likes(John, x)$
b. $food(Apple) \wedge food(Vegetables)$
c. $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f. $\forall x : \neg killed(x) \rightarrow alive(x)$
g. $\forall x : alive(x) \rightarrow \neg killed(x)$
h. $likes(John, Peanuts)$

→ Eliminate implication

$\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(Vegetable)$
c. $\forall x \forall y \neg [eats(x, y) \wedge \neg killed(x)] \vee food(y)$
d. $eats(Anil, peanuts) \wedge alive(Anil)$
e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f. $\forall x \neg [\neg killed(x)] \vee alive(x)$
g. $\forall x \neg alive(x) \vee \neg killed(x)$
h. $likes(John, Peanuts)$

→ Move negation ($\neg$) inwards & rewrite

a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(Vegetables)$
c. $\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f. $\forall x \neg killed(x) \vee alive(x)$
g. $\forall x \neg alive(x) \vee \neg killed(x)$
h. $likes(John, Peanuts)$

→ Rename variables

a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(Vegetables)$
c. $\forall x \forall y \neg eats(y, z) \vee killed(y) \vee food(z)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$
f. $\forall g [killed(g)] \vee alive(g)$
g. $\forall k \neg alive(k) \vee \neg killed(k)$
h. $likes(John, Peanuts)$

→ Drop Universal Quantifiers

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

$\neg likes(John, Peanuts)$ — $\neg food(x) \vee likes(John, x)$ {Peanuts/x}
$\neg food(Peanuts)$ — $\neg eats(y, z) \vee killed(y) \vee food(z)$ {Peanuts/z}
$\neg eats(y, Peanuts) \vee killed(y)$ — $eats(Anil, Peanuts)$ {Anil/y}
$killed(Anil)$ — $\neg alive(k) \vee \neg killed(k)$ {Anil/k}
$\neg alive(Anil)$ — $alive(Anil)$
{}
Hence Proved

Code-

```
# -*- coding: utf-8 -*-
from copy import deepcopy

U = {
   'a': "∀x: food(x) → likes(John, x)",  'b':
   "food(Apple) ∧ food(Vegetables)",
   'c': "∀x∀y: eats(x, y) ∧ ¬killed(x) → food(y)",
   'd': "eats(Anil, Peanuts) ∧ alive(Anil)",
   'e': "∀x: eats(Anil, x) → eats(Harry, x)",
   'f': "∀x: ¬killed(x) → alive(x)",
   'g': "∀x: alive(x) → ¬killed(x)",
   'h': "likes(John, Peanuts)"
}

print("=== STEP 1: Given FOL Statements ===")
for k, v in U.items():
   print(f"{k}. {v}")

print("\n=== STEP 2: After Removing Implications ===")

S1 = {
   'a': "¬food(x) ∨ likes(John, x)",
   'b1': "food(Apple)",
   'b2': "food(Vegetables)",
   'c': "¬eats(x, y) ∨ killed(x) ∨ food(y)",
   'd1': "eats(Anil, Peanuts)",
   'd2': "alive(Anil)",
   'e': "¬eats(Anil, x) ∨ eats(Harry, x)",
   'f': "killed(x) ∨ alive(x)",
   'g': "¬alive(x) ∨ ¬killed(x)",
   'h': "likes(John, Peanuts)"
}

for k, v in S1.items():
   print(f"{k}. {v}")

print("\n=== STEP 3: Standardized Variables (Dropped Quantifiers) ===")
for k, v in S1.items():
   print(f"{k}. {v}")

print("\n=== STEP 4: Final CNF Clauses ===")

KB_cnf = [
   "¬food(x) ∨ likes(John, x)",
   "food(Apple)",
```

```
    "food(Vegetables)",
    "¬eats(y, z) ∨ killed(y) ∨ food(z)",
    "eats(Anil, Peanuts)",
    "alive(Anil)",
    "¬eats(Anil, w) ∨ eats(Harry, w)",
    "killed(g) ∨  alive(g)",
    "¬alive(k) ∨ ¬killed(k)",
    "likes(John, Peanuts)"
]

for i, c in enumerate(KB_cnf, 1):
    print(f"{i}. {c}")

print("\n=== STEP 5: Resolution Proof ===")

proof = [
    ("1", "Negate the Goal", "¬likes(John, Peanuts)"),
    ("2", "Resolve (1) with (¬food(x) ∨ likes(John, x)) using {x/Peanuts}",
"¬food(Peanuts)"),
    ("3", "Resolve (2) with (¬eats(y,z) ∨ killed(y) ∨ food(z)) using {z/Peanuts}",
"¬eats(y,Peanuts) ∨  killed(y)"),
    ("4", "Resolve (3) with (eats(Anil, Peanuts)) using {y/Anil}", "killed(Anil)"),
    ("5", "Resolve (4) with (¬alive(k) ∨ ¬killed(k)) using {k/Anil}",
"¬alive(Anil)"),
    ("6", "Resolve (5) with (alive(Anil))", "⊥ (Contradiction)")
]

for step_id, desc, out in proof:
    print(f"Step {step_id}: {desc}")
    print(f"      ⇒ {out}\n")

print("Contradiction reached ⇒ Therefore, John likes Peanuts is TRUE.\n")
```

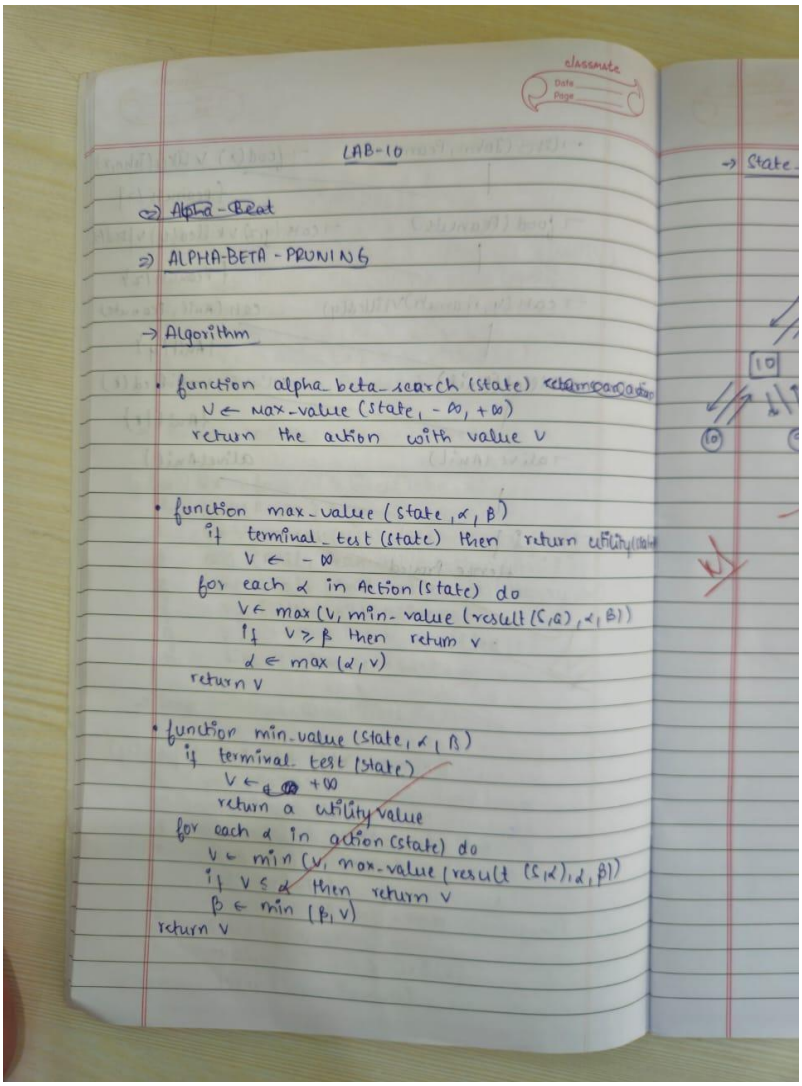
```
=== STEP 1: Given FOL Statements ===
a. ∀x: food(x) → likes(John, x)
b. food(Apple) ∧ food(Vegetables)
c. ∀x∀y: eats(x, y) ∧ ¬killed(x) → food(y)
d. eats(Anil, Peanuts) ∧ alive(Anil)
e. ∀x: eats(Anil, x) → eats(Harry, x)
f. ∀x: ¬killed(x) → alive(x)
g. ∀x: alive(x) → ¬killed(x)
h. likes(John, Peanuts)

=== STEP 2: After Removing Implications ===
a. ¬food(x) ∨ likes(John, x)
b1. food(Apple)
b2. food(Vegetables)
c. ¬eats(x, y) ∨ killed(x) ∨ food(y)
d1. eats(Anil, Peanuts)
d2. alive(Anil)
e. ¬eats(Anil, x) ∨ eats(Harry, x)
f. killed(x) ∨ alive(x)
g. ¬alive(x) ∨ ¬killed(x)
h. likes(John, Peanuts)

=== STEP 3: Standardized Variables (Dropped Quantifiers) ===
a. ¬food(x) ∨ likes(John, x)
b1. food(Apple)
b2. food(Vegetables)
c. ¬eats(x, y) ∨ killed(x) ∨ food(y)
d1. eats(Anil, Peanuts)
d2. alive(Anil)
e. ¬eats(Anil, x) ∨ eats(Harry, x)
f. killed(x) ∨ alive(x)
g. ¬alive(x) ∨ ¬killed(x)
h. likes(John, Peanuts)
```

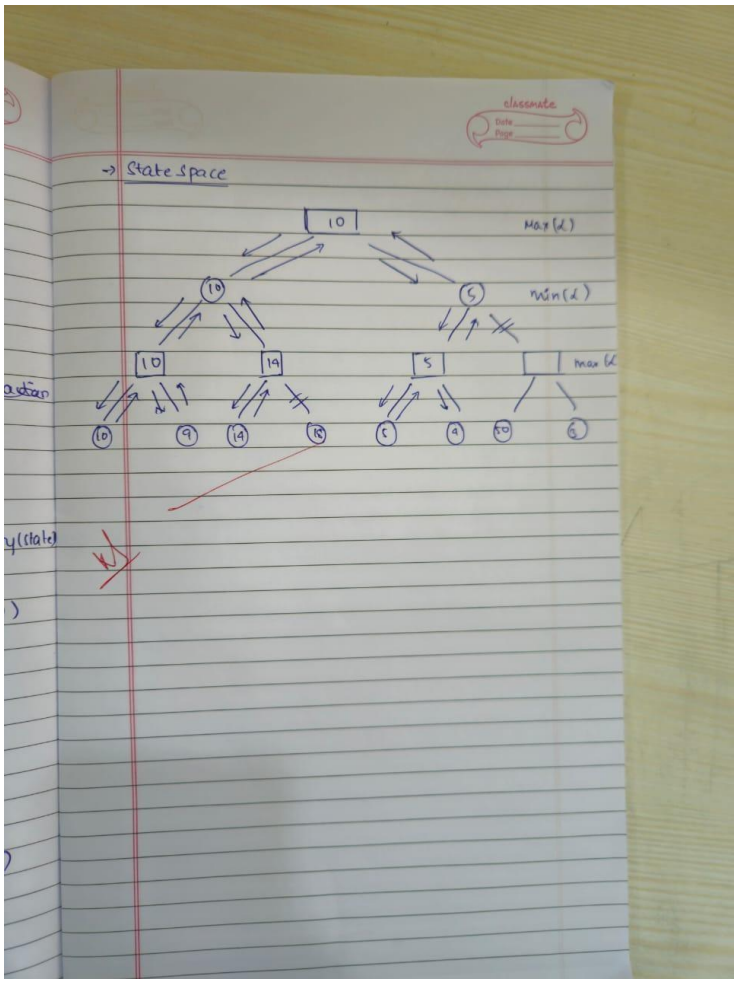
```
=== STEP 4: Final CNF Clauses ===
1. ¬food(x) ∨ likes(John, x)
2. food(Apple)
3. food(Vegetables)
4. ¬eats(y, z) ∨ killed(y) ∨ food(z)
5. eats(Anil, Peanuts)
6. alive(Anil)
7. ¬eats(Anil, w) ∨ eats(Harry, w)
8. killed(g) ∨ alive(g)
9. ¬alive(k) ∨ ¬killed(k)
10. likes(John, Peanuts)

=== STEP 5: Resolution Proof ===
Step 1: Negate the Goal
       ⇒ ¬likes(John, Peanuts)

Step 2: Resolve (1) with (¬food(x) ∨ likes(John, x)) using {x/Peanuts}
       ⇒ ¬food(Peanuts)

Step 3: Resolve (2) with (¬eats(y,z) ∨ killed(y) ∨ food(z)) using {z/Peanuts}
       ⇒ ¬eats(y,Peanuts) ∨ killed(y)

Step 4: Resolve (3) with (eats(Anil, Peanuts)) using {y/Anil}
       ⇒ killed(Anil)

Step 5: Resolve (4) with (¬alive(k) ∨ ¬killed(k)) using {k/Anil}
       ⇒ ¬alive(Anil)

Step 6: Resolve (5) with (alive(Anil))
       ⇒ ⊥ (Contradiction)

Contradiction reached ⇒ Therefore, John likes Peanuts is TRUE.
```

Program 10-

Implement Alpha-Beta Pruning.

ALGORITHM-





Code-

```
import math

T = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['L1', 'L2'],
    'E': ['L3', 'L4'],
    'F': ['L5', 'L6'],
    'G': ['L7', 'L8'],
    'L1': 10, 'L2': 9, 'L3': 14, 'L4': 18,
    'L5': 5,  'L6': 4, 'L7': 50, 'L8': 3
}

def banner():
    print("\nGame Tree Structure:\n")
    print("                A [MAX]")
    print("              /       \\")
    print("           B [MIN]       C [MIN]")
    print("          /    \\       /    \\")
    print("       D [MAX]  E [MAX]  F [MAX]  G [MAX]")
    print("      /  \\     /  \\     /  \\     /  \\")
    print("    10    9  14   18  5     4  50    3")
    print("\n ---------------------------------------- \n")

def ab(node, depth, a, b, maximizing):
    pad = "  " * depth
    if isinstance(T[node], int):
        print(f"{pad}Leaf {node} -> {T[node]}")
        return T[node]

    if maximizing:
        print(f"{pad}MAX {node} @ depth={depth}, α={a}, β={b}")
        best = -math.inf
        for ch in T[node]:
            print(f"{pad}→ visit {ch}")
            val = ab(ch, depth + 1, a, b, False)
            best = max(best, val)
            a = max(a, val)
            print(f"{pad}MAX update {node}: best={best}, α={a}, β={b}")
            if b <= a:
                print(f"{pad}PRUNE at MAX {node} (β={b} ≤ α={a})")
                break
        return best
    else:
```

```
        print(f"{pad}MIN {node} @ depth={depth}, α={a}, β={b}")
        best = math.inf
        for ch in T[node]:
            print(f"{pad}→ visit {ch}")
            val = ab(ch, depth + 1, a, b, True)
            best = min(best, val)
            b = min(b, val)
            print(f"{pad}MIN update {node}: best={best}, α={a}, β={b}")
            if b <= a:
                print(f"{pad}PRUNE at MIN {node} (β={b} ≤ α={a})")
                break
        return best

banner()
print("Starting Alpha-Beta Pruning...\n")
ans = ab('A', 0, -math.inf, math.inf, True)
print("\n ---------------------------------------- ")
print(f"☑ Best achievable value at root (A):
{ans}") print("---------------------------------- ")
```

```
Game Tree Structure:

                A [MAX]
              /         \
          B [MIN]       C [MIN]
         /       \      /      \
      D [MAX]  E [MAX]  F [MAX]  G [MAX]
     /   \     /   \    /   \    /   \
    10    9   14   18   5    4   50    3

----------------------------------------

Starting Alpha-Beta Pruning...

MAX A @ depth=0, α=-inf, β=inf
→ visit B
  MIN B @ depth=1, α=-inf, β=inf
  → visit D
    MAX D @ depth=2, α=-inf, β=inf
    → visit L1
      Leaf L1 -> 10
    MAX update D: best=10, α=10, β=inf
    → visit L2
      Leaf L2 -> 9
    MAX update D: best=10, α=10, β=inf
  MIN update B: best=10, α=-inf, β=10
  → visit E
    MAX E @ depth=2, α=-inf, β=10
    → visit L3
      Leaf L3 -> 14
    MAX update E: best=14, α=14, β=10
    PRUNE at MAX E (β=10 ≤ α=14)
  MIN update B: best=10, α=-inf, β=10
MAX update A: best=10, α=10, β=inf
→ visit C
  MIN C @ depth=1, α=10, β=inf
  → visit F
    MAX F @ depth=2, α=10, β=inf
    → visit L5
      Leaf L5 -> 5
    MAX update F: best=5, α=10, β=inf
    → visit L6
      Leaf L6 -> 4
```

```
→ visit C
  MIN C @ depth=1, α=10, β=inf
  → visit F
    MAX F @ depth=2, α=10, β=inf
    → visit L5
      Leaf L5 -> 5
    MAX update F: best=5, α=10, β=inf
    → visit L6
      Leaf L6 -> 4
    MAX update F: best=5, α=10, β=inf
  MIN update C: best=5, α=10, β=5
  PRUNE at MIN C (β=5 ≤ α=10)
MAX update A: best=10, α=10, β=inf

------------------------------------------
✅ Best achievable value at root (A): 10
------------------------------------------
```