VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Jayashree Tarai (1BM24CS407)

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 **Jayashree Tarai (1BM24CS407),** 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

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

Program-01

Implement Tic Tac Toe Game

Experiment 01 21/8/25

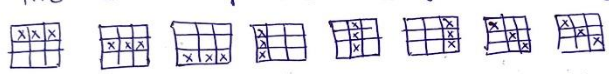
Problem Statement

Design and implement a Tic-Tac-Toe game Using arrays the players can be either both computer vs Computer, Computer Vs human

Logic

→ The board is 3x3 grid Stored in 2D Array Empty cells are marked with -

→ The 8 winning Conditions (for X or O)



→ If any of these conditions are true then the player 1 or player 2 wins

→ If the board is full and no ~~has~~ one has these Condition then its Draw

Computer VS Human

→ Show the board, If its a human take input of that partiular row & column (X)

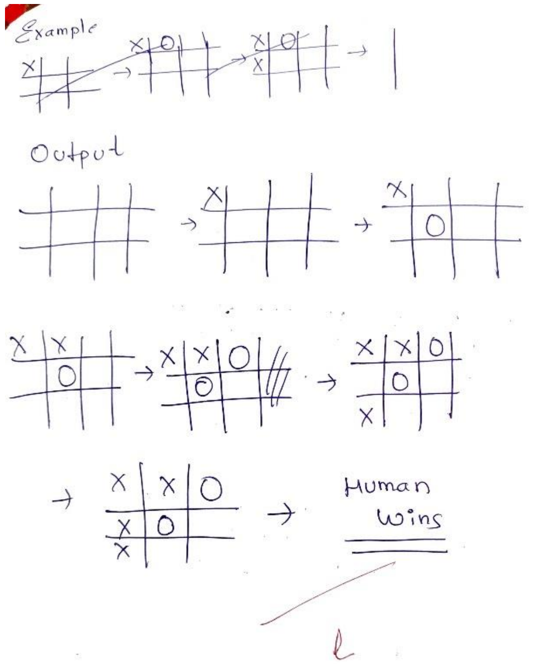
→ Write a function for the Computer to place the Sign in an Empty cell ↓ to place (O)

→ Check for the Winner Conditions If any one of the player meets the 8 winning conditions then declare the player as Winner (8: 3 rows, 3 columns, 2 Diagonal Conditions)

→ If None of them are meeting the winning Condition then Declare ~~the~~ Draw

21/8/2

Algorithm:



Code:

```
import random

grid = []
line = []
for i in range(3):
    for j in range(3):
        line.append(" ")
    grid.append(line)
    line = []

# grid printing
def print_grid():
    for i in range(3):
        print("|", end="")
```

```
        for j in range(3):
            print(grid[i][j], "|", end="")
        print("")

# player turn
def player_turn(turn_player1):
    if turn_player1 == True:
        turn_player1 = False
        print(f"It's {player2}'s turn")
    else:
        turn_player1 = True
        print(f"It's {player1}'s turn")
    return turn_player1

# choosing cell
def write_cell(cell, turn_player1):
    cell -= 1
    i = int(cell / 3)
    j = cell % 3
    if turn_player1 == True:
        grid[i][j] = player1_symbol
    else:
        grid[i][j] = player2_symbol
    return grid

# checking if cell is free
def free_cell(cell):
    cell -= 1
    i = int(cell / 3)
    j = cell % 3
    if grid[i][j] == player1_symbol or grid[i][j] == player2_symbol:
        print("This cell is not free")
        return False
    return True

# system turn (AI)
def system_turn():
```

```
    empty_cells = [i for i in range(1, 10) if free_cell(i)]
    if empty_cells:
        return random.choice(empty_cells)
    return None

# win check
def win_check(grid, player1_symbol, player2_symbol):
    full_grid = True
    player1_symbol_count = 0
    player2_symbol_count = 0
    # checking rows
    for i in range(3):
        for j in range(3):
            if grid[i][j] == player1_symbol:
                player1_symbol_count += 1
                player2_symbol_count = 0
                if player1_symbol_count == 3:
                    game = False
                    winner = player1
                    return game, winner
            if grid[i][j] == player2_symbol:
                player2_symbol_count += 1
                player1_symbol_count = 0
                if player2_symbol_count == 3:
                    game = False
                    winner = player2
                    return game, winner
            if grid[i][j] == " ":
                full_grid = False

        player1_symbol_count = 0
        player2_symbol_count = 0
    # checking columns
    player1_symbol_count = 0
    player2_symbol_count = 0
    for i in range(3):
        for j in range(3):
```

```
            for k in range(3):
                if i + k <= 2:
                    if grid[i + k][j] == player1_symbol:
                        player1_symbol_count += 1
                        player2_symbol_count = 0
                        if player1_symbol_count == 3:
                            game = False
                            winner = player1
                            return game, winner
                    if grid[i + k][j] == player2_symbol:
                        player2_symbol_count += 1
                        player1_symbol_count = 0
                        if player2_symbol_count == 3:
                            game = False
                            winner = player2
                            return game, winner
            if grid[i][j] == " ":
                full_grid = False

            player1_symbol_count = 0
            player2_symbol_count = 0
    # checking diagonals
    player1_symbol_count = 0
    player2_symbol_count = 0
    for i in range(3):
        for j in range(3):
            for k in range(3):
                if j + k <= 2 and i + k <= 2:
                    if grid[i + k][j + k] == player1_symbol:
                        player1_symbol_count += 1
                        player2_symbol_count = 0
                        if player1_symbol_count == 3:
                            game = False
                            winner = player1
                            return game, winner
                    if grid[i + k][j + k] == player2_symbol:
                        player2_symbol_count += 1
```

```
                player1_symbol_count = 0
                if player2_symbol_count == 3:
                    game = False
                    winner = player2
                    return game, winner
        if grid[i][j] == " ":
            full_grid = False

        player1_symbol_count = 0
        player2_symbol_count = 0

player1_symbol_count = 0
player2_symbol_count = 0
for i in range(3):
    for j in range(3):
        for k in range(3):
            if j - k >= 0 and i + k <= 2:
                if grid[i + k][j - k] == player1_symbol:
                    player1_symbol_count += 1
                    player2_symbol_count = 0
                    if player1_symbol_count == 3:
                        game = False
                        winner = player1
                        return game, winner
                if grid[i + k][j - k] == player2_symbol:
                    player2_symbol_count += 1
                    player1_symbol_count = 0
                    if player2_symbol_count == 3:
                        game = False
                        winner = player2
                        return game, winner
        if grid[i][j] == " ":
            full_grid = False

        player1_symbol_count = 0
        player2_symbol_count = 0
```

```
    # full grid or not
    if full_grid == True:
        game = False
        winner = ""
        return game, winner
    else:
        game = True
        winner = ""
        return game, winner

# game mode selection
def game_mode_selection():
    print("Choose the game mode:")
    print("1. User vs User")
    print("2. User vs System")
    choice = input("Enter 1 or 2: ")
    if choice == "1":
        return "User vs User"
    elif choice == "2":
        return "User vs System"
    else:
        print("Invalid choice! Please enter 1 or 2.")
        return game_mode_selection()

# game opening
print("Welcome to Tic-Tac-Toe!")
mode = game_mode_selection()
print("")

# input player names and symbols
player1 = input("Please enter name of player 1: ")
player1_symbol = input(f"Please enter the symbol for {player1}: ")

if mode == "User vs User":
    player2 = input("Please enter name of player 2: ")
    player2_symbol = input(f"Please enter the symbol for {player2}: ")
else:
```

```
    player2 = "System"
    player2_symbol = "O" if player1_symbol == "X" else "X"  # Automatically set to the opposite symbol
of player 1

game = True
full_grid = False
turn_player1 = True
winner = ""

# game loop
while game:
    turn_player1 = player_turn(turn_player1)
    free_box = False
    while free_box == False:
        if turn_player1:  # Player 1's turn
            cell = int(input(f"{player1}, enter a number (1 to 9): "))
        else:
            if mode == "User vs System":  # System's turn
                print(f"{player2}'s turn (System)")
                cell = system_turn()
                print(f"System chose cell {cell}")
            else:  # Player 2's turn (User vs User)
                cell = int(input(f"{player2}, enter a number (1 to 9): "))

        free_box = free_cell(cell)
    grid = write_cell(cell, turn_player1)
    print_grid()

    game, winner = win_check(grid, player1_symbol, player2_symbol)

# end of game
if winner == player1:
    print(f"Winner is {player1}!")
elif winner == player2:
    print(f"Winner is {player2}!")
else:
    print("It's a draw!")
```

Output:

```
Welcome to Tic-Tac-Toe!
Choose the game mode:
1. User vs User
2. User vs System
Enter 1 or 2: 1

Please enter name of player 1: dhanush
Please enter the symbol for dhanush: x
Please enter name of player 2: srujan
Please enter the symbol for srujan: o
It's srujan's turn
srujan, enter a number (1 to 9): 5
|   |   |   |
|   |o  |   |
|   |   |   |
It's dhanush's turn
dhanush, enter a number (1 to 9): 6
|   |   |   |
|   |o  |x  |
|   |   |   |
It's srujan's turn
srujan, enter a number (1 to 9): 7
|   |   |   |
|   |o  |x  |
|o  |   |   |
It's dhanush's turn
dhanush, enter a number (1 to 9): 3
|   |   |x  |
|   |o  |x  |
|o  |   |   |
It's srujan's turn
srujan, enter a number (1 to 9): 9
|   |   |x  |
|   |o  |x  |
|o  |   |o  |
It's dhanush's turn
dhanush, enter a number (1 to 9):
=== Session Ended. Please Run the code again ===
```

Program-02

Implement Vacuum Cleaner

Algorithm:

Vaccum cleaner Agent

Problem statement:

Design an Algorithm where there are 4 rooms and the Vaccum cleaner should clean the room if its Dirty $2^4 = 8$

$n \times n^{la} = 4 \times 2^4$

Step1: start the agent in any one of the 4rooms (A, B or C, D)

Step2: check if the Current room is Dirty or clean

Step3: If the room is Dirty then perform action Suck (clean the room)

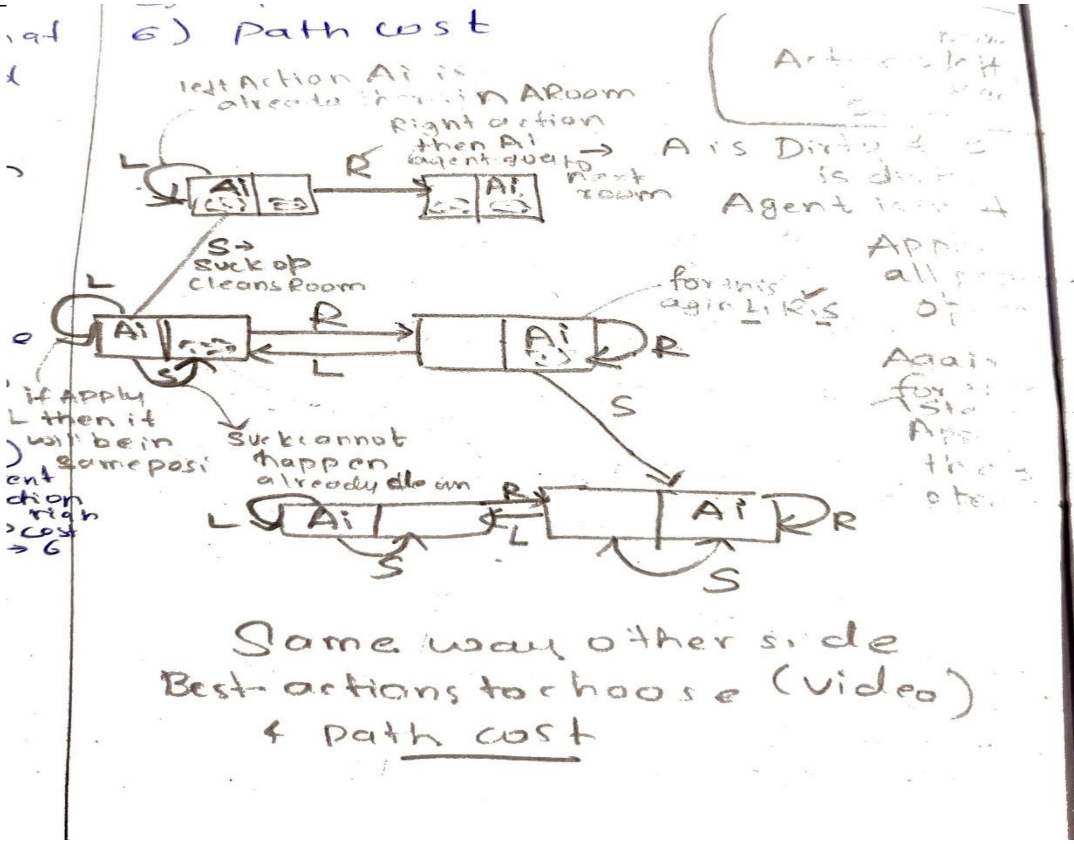
Step4: If the room is clean Move to the next room /or Right

If in A → B or B → C, C → D, D → A

Step 5: Repeat 2-4 steps until all rooms are Clean

Step 6: stop the process

```
Initialize state = {"A": Unknown, "B": Unknown, "C":
  Unknown, "D": Unknown}
Path = ["A", "B", "C", "D"] Current-index = index of starting room
function VaccumAgent (location, status):
  State [location] = status
if status == "Dirty":
  return "Suck"
if all room in state are "clean":
  return "Do Nothing"
if Current-index <3:
  Current-index += 1
  return "Move Right"
else if current-index >0
  current-index -= 1
  return "Move Left"
```



Code:

```
# Helper function: Check if all rooms are clean
def all_rooms_clean(environment):
    return all(status == "clean" for status in environment.values())

# Helper function: Clean a specific room
def clean_room(environment, room):
    print(f"Cleaning {room}")
    environment[room] = "clean"

# Helper function: Move to next room in the 2x2 grid
def move_to_next_room(current_room):
    # Predefined cyclic movement order
    if current_room == "room1":
        return "room2"
    elif current_room == "room2":
        return "room4"
    elif current_room == "room4":
        return "room3"
    elif current_room == "room3":
```

```
return "room1"
```

```
# Vacuum cleaner agent function
def vacuum_cleaner_agent(environment):
    current_room = "room1"
    steps = 0

    print("Initial environment:", environment)
    print(f"Starting cleaning in {current_room}\n")

    while not all_rooms_clean(environment):
        if environment[current_room] == "dirty":
            clean_room(environment, current_room)
        else:
            print(f"{current_room} is already clean, moving on.")

        current_room = move_to_next_room(current_room)
        steps += 1
        print(f"Moved to {current_room}\n")

    print("All rooms cleaned!")
    print("Final environment:", environment)
    print(f"Total steps taken: {steps}")

# Initialize environment with all rooms dirty
environment = {
    "room1": "dirty",
    "room2": "dirty",
    "room3": "dirty",
    "room4": "dirty"
}

# Run the agent
vacuum_cleaner_agent(environment)
```

Output:

```
Vacuum Cleaner World
Commands: LEFT, RIGHT, CLEAN, EXIT

[R:dirty] [B:dirty]

Enter command: left
Already at the leftmost room!
[R:dirty] [B:dirty]

Enter command: right
[A:dirty] [R:dirty]

Enter command: clean
Cleaned B
[A:dirty] [R:clean]

Enter command: left
[R:dirty] [B:clean]

Enter command: clean
Cleaned A
[R:clean] [B:clean]

🏆 All rooms are clean!
```

Program-03

Implement 8 Puzzle

Algorithm:

8 Puzzle Game

Algorithm

8 Puzzle Game

→ Contains 3x3 grid 9 cells

→ one of the cell is blank cell & the remaining

→ will have from 1-8 with shuffled

→ See the first rows like how many cells are Correct & which is in the wrong cell

→ Try to move the blank space up, down, left right so that you can get the first row

goal 1 2 3

→ While the Queue is not empty

• Remove the first state from the Queue (this is current state you're exploring)

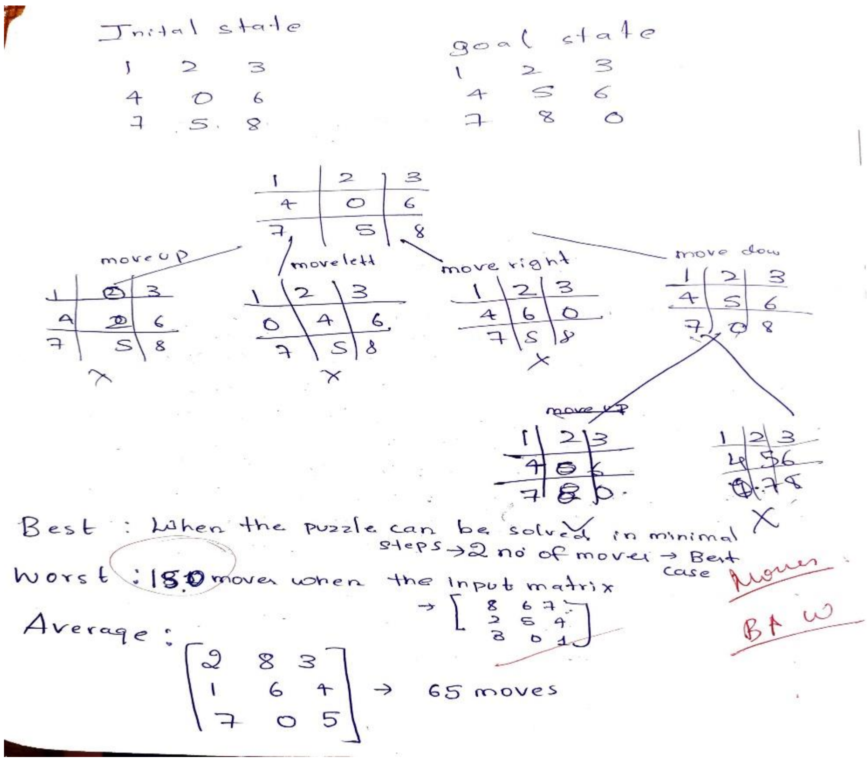
• Check if the current state is the goal done if it is we're done the soulution is found

→ If the current state isn't the goal generate all possible valid states from this state by moving up the empty space (left, right)

• Check if it has new state generated check if has not been visited add it

→ Repeat the procen until

→ End if the goal state is finished the algorithm terminates



Code:

```
import heapq

# Goal state (target configuration)
GOAL_STATE = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

# Manhattan Distance Heuristic
def manhattan(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            val = state[i][j]
            if val == 0:
                continue
            goal_x = (val - 1) // 3
            goal_y = (val - 1) % 3
```

```
            distance += abs(goal_x - i) + abs(goal_y - j)
    return distance

# Check if the current state is the goal state
def is_goal(state):
    return state == GOAL_STATE

# Find neighbors (possible moves from current state)
def get_neighbors(state):
    neighbors = []
    # Find the position of the blank tile (0)
    x, y = [(ix, iy) for ix, row in enumerate(state) for iy, i in enumerate(row) if i ==
0][0]
    # Possible moves for the blank tile: up, down, left, right
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    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]  # Create a new state
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
# Swap blank with neighbor
            neighbors.append(new_state)
    return neighbors

# A* search algorithm to solve the puzzle
def solve_puzzle(start):
    # Priority queue to store the states (open set)
    heap = []
    # Push initial state into the priority queue with f(n) = g(n) + h(n)
    heapq.heappush(heap, (manhattan(start), 0, start, [])) # f(n), g(n), state, path to
get there
    visited = set()  # Set to store visited states to avoid reprocessing

    while heap:
        est_total, cost, state, path = heapq.heappop(heap)  # Pop the state with
lowest f(n)
        key = str(state)
```

```
        # Skip the state if it has already been visited
        if key in visited:
            continue
        visited.add(key)

        # If we reached the goal state, return the solution path
        if is_goal(state):
            return path + [state]

        # Explore the neighbors
        for neighbor in get_neighbors(state):
            # Push each neighbor to the priority queue
            heapq.heappush(heap, (cost + 1 + manhattan(neighbor), cost + 1,
neighbor, path + [state]))

    # If no solution found
    return None



def print_state(state):
    for row in state:
        print(row)
    print()
initial_state = [
    [1, 2, 3],
    [4, 0, 6],
    [7, 5, 8]
]

print("Initial State:")
print_state(initial_state)
solution = solve_puzzle(initial_state)
if solution:
    print("Solution found in {} steps:".format(len(solution) - 1))
    for step, state in enumerate(solution):
        print(f"Step {step}:")
```

```
        print_state(state)
else:
    print("No solution found.")
```

Output:

```
Output
Initial State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Solution found in 2 steps:
Step 0:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 1:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Program-04

Implement IDDFS

Algorithm:

8 Puzzle Using misplaced tiles & Manhattan & IDDFS

→ 8 Puzzle Using IDDFS

IDDFS Algorithm (Iterative Deepening Depth first Search)

→ IDDFS is an Uninformed Search algorithm (iterative approach)

→ combines both DFS(memory efficiency) & BFS(Completeness & optimality)

→ Effective for large graphs where the depth is Unknown

Algorithm

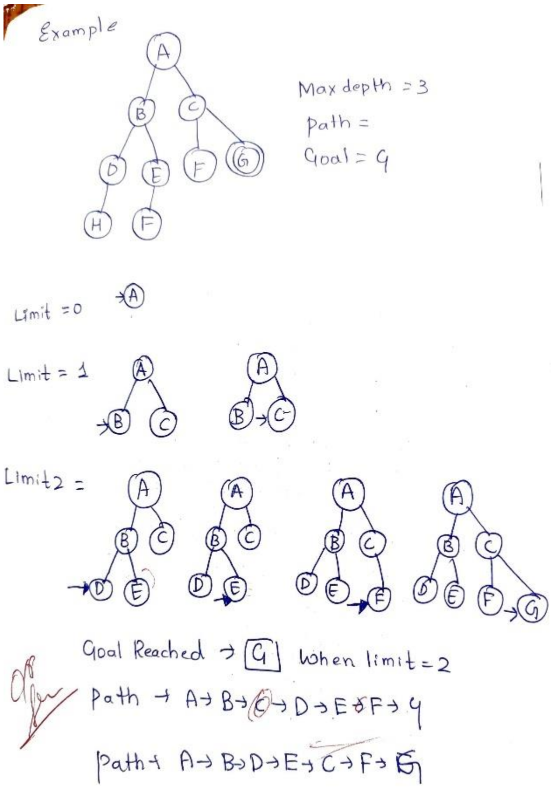
step1: ·The initial depth limit =0

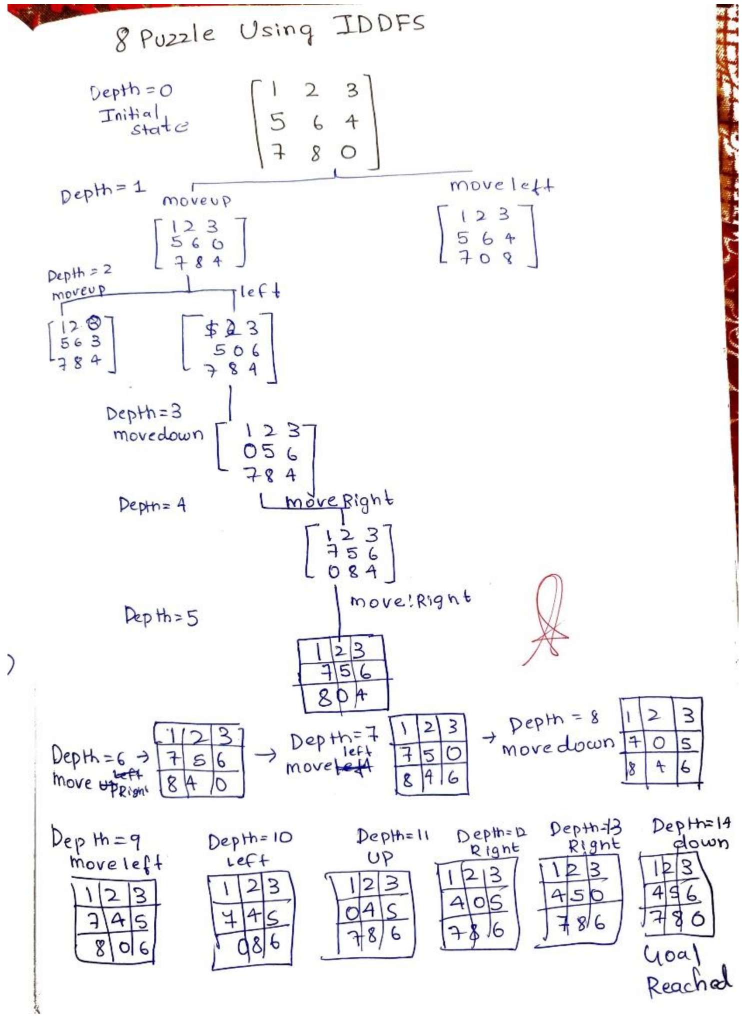
step2: Perform Depth limited Depth first Search from starting node

step3: If the goal node is encountered during the search then Return the path from the start node to goal node within the current depth limit

Step 4: If the goal node is not found within current depth limit increase the depth limit by 1 and Repeat steps 2 & 3

Step5: If the goal node is not found & the depth limit exceeds a predefined maximum depth ~~the~~ or the depth of tree then terminate the Process.





Code:

```
from collections import deque


moves = {
```

```
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

def get_neighbors(state):
    neighbors = []
    zero_idx = state.index("0")

    for move, pos_change in moves.items():
        new_idx = zero_idx + pos_change


        if move == 'L' and zero_idx % 3 == 0:
            continue
        if move == 'R' and zero_idx % 3 == 2:
            continue
        if 0 <= new_idx < 9:
            new_state = list(state)

            new_state[zero_idx], new_state[new_idx] = new_state[new_idx], new_state[zero_idx]
            neighbors.append(("".join(new_state), move))
    return neighbors


def depth_limited_search(state, goal, limit, path, visited):
    if state == goal:
        return path

    if limit <= 0:
        return None

    visited.add(state)
    for neighbor, move in get_neighbors(state):
        if neighbor not in visited:
            new_path = depth_limited_search(neighbor, goal, limit - 1, path + [move], visited.copy())
            if new_path:
                return new_path
    return None
def iterative_deepening_search(start, goal, max_depth=30):
    for depth in range(max_depth + 1):
        print(f"\n🔄 Searching with depth limit = {depth}")
        path = depth_limited_search(start, goal, depth, [], set())
        if path:
            print(f"✅Goal found at depth {depth}! Moves: {' -> '.join(path)}")
```

```
            return path
    print(f"+ Goal not found within depth {max_depth}")
    return None



start_state = "724506831"
goal_state  = "012345678"

iterative_deepening_search(start_state,  goal_state,  max_depth=20)
```

Output:

```
🔎 Searching with depth limit = 0
🔎 Searching with depth limit = 1
🔎 Searching with depth limit = 2
🔎 Searching with depth limit = 3
🔎 Searching with depth limit = 4
🔎 Searching with depth limit = 5
🔎 Searching with depth limit = 6
🔎 Searching with depth limit = 7
🔎 Searching with depth limit = 8
🔎 Searching with depth limit = 9
🔎 Searching with depth limit = 10
🔎 Searching with depth limit = 11
🔎 Searching with depth limit = 12
```

Program-05

Implement Hill climbing algorithm using N- Queens

Hill climbing

- To find an optimal (near-optimal) solution by iteratively moving from current state to a better neighbouring state accordingly to heuristic function

Algorithm

Step 1 : Initial state
start, Begin with the start state / choose the random state

Step 2 : Neighbouring state
Identify the neighbouring states of that current solution by making small adjustment (mutation

Step 3 : Move to Neigbour
If ~~the~~ any of the neighbouring state offers like better solution Using some heuristic function then leave the current state & move to that new state

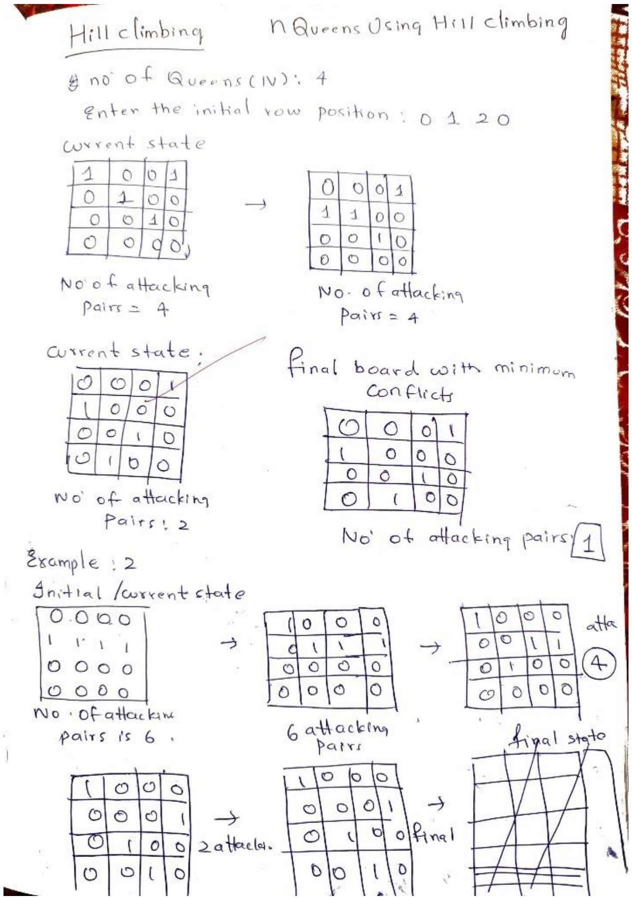
Step 4 : Termination (stops when no better neighborr exit)
Repeat this process until no neighbouring state has the better solution than current one

Pseudocode

Function Hillclimbing (problem) :
current_state = start_state (problem)
loop (forever) :
neighbour/solution = ~~higher~~ Generated-neighbour currentstate

If neighbour is empty:
Return current state
best neighbour = Select Best solution/Neigbour
currentstate, problem (Evaluate problem)

Algorithm:



Code:

```
from random import randint

def configureUserInput(board, state, N):
    while True:
        try:
            initial_positions = input(f"Enter the initial row positions for queens
(space-separated, 0 to {N-1}): ")
            initial_positions = list(map(int, initial_positions.split()))
            if len(initial_positions) == N and all(0 <= x < N for x in
initial_positions):
                for i in range(N):
                    state[i] = initial_positions[i]
```

```
                board[state[i]][i] = 1
                break
            else:
                print(f"Please enter exactly {N} valid row positions between 0 and
{N-1}.")
        except ValueError:
            print("Invalid input. Please enter integers.")

def printBoard(board):
    for row in board:
        print(*row)

def compareStates(state1, state2):
    return state1 == state2

def fill(board, value):
    for i in range(len(board)):
        for j in range(len(board)):
            board[i][j] = value

def calculateObjective(board, state, N):
    attacking = 0
    for i in range(N):
        row = state[i]
        col = i
        for j in range(i+1, N):
            other_row = state[j]
            other_col = j
            if other_row == row or abs(other_row - row) == abs(other_col - col):
                attacking += 1
    return attacking

def generateBoard(board, state, N):
    fill(board, 0)
    for i in range(N):
        board[state[i]][i] = 1
```

```
def copyState(state1, state2):
    for i in range(len(state2)):
        state1[i] = state2[i]

def getNeighbour(board, state, N):
    opState = state[:]
    generateBoard(board, opState, N)
    opObjective = calculateObjective(board, opState, N)

    for col in range(N):
        for row in range(N):
            if row != state[col]:
                tempState = state[:]
                tempState[col] = row
                generateBoard(board, tempState, N)
                tempObjective = calculateObjective(board, tempState, N)
                if tempObjective < opObjective:
                    opObjective = tempObjective
                    opState = tempState[:]
    copyState(state, opState)
    generateBoard(board, state, N)

def hillClimbing(board, state, N):
    neighbourState = state[:]
    generateBoard(board, neighbourState, N)

    while True:
        print("\nCurrent state:")
        printBoard(board)
        print(f"Number of attacking pairs: {calculateObjective(board, state, N)}")

        copyState(state, neighbourState)
        generateBoard(board, state, N)

        getNeighbour(board, neighbourState, N)

        if compareStates(state, neighbourState):
```

```
            print("\nFinal board with minimum conflicts:")
            printBoard(board)
            print(f"Number of attacking pairs: {calculateObjective(board, state,
N)}")
            break

def main():
    N = int(input("Enter number of queens (N): "))
    board = [[0 for _ in range(N)] for _ in range(N)]
    state = [0] * N

    print("Enter initial positions for queens ")
    configureUserInput(board, state, N)

    hillClimbing(board, state, N)

if _name___== "_main_":
    main()
```

Output:

```
Initial board:
. . . .
. Q . .
. . . .
Q . Q Q

Initial heuristic (attacking pairs): 4

Step 0: h = 4
. . . .
. Q . .
. . . .
Q . Q Q

Step 1: h = 2
. . . Q
. Q . .
. . . .
Q . Q .

Step 2: h = 1
. . . Q
. Q . .
Q . . .
. . Q .

Final board:
. . . Q
. Q . .
Q . . .
. . Q .

Final heuristic: 1
⚠ Local minimum reached (no solution from this start).
```

Program-06

Implement Simulated Annealing

Simulated Annealing

is an optimization algorithm designed to search for an optimal solution in Vast solution space & can also move to worse neighbour

Algorithm

1) Start with the initial solution current
2) set an initial temperature t
3) Repeat until t ~~has~~ is close to 0
4) ~~if near_solution~~ Generate a new solution
5) If the new solution is better then Accept
6) If worse then accept it with some probability
7) Decrease the temperature

Pseudocode

$P = e^{\frac{\Delta E}{T}}$

Delta = e Δ

```
function Simulated Annealing():
    Current = Initial_solution()
    T = initial temperature
    best = current
    while T > min_temperature
        new_sol = get neighbour (current)
        cost_diff = cost (new_sol)

        If cost_diff < 0 Then
            curr_sol = curr neighbour
            curr_cost ← neighborCost
        else
            probabiti ← exp (-delta / temp)
            currsol ← neigh
            currentcost ← neigbour cost
```

Algorithm:

Simulated Annealing

Initial board

```
. . . .
Q . . Q
. Q . .
. . Q .
```

→

Iteration 0
temp: 100.00
Attempting ~~row~~ to move
Queen in col 0 from
row 1 to row 2
now every 2 is down

Conflict: 4

Iteration 1

```
. . . .
. . . Q
Q Q . .
. . Q .
```

conflict 2

Iteration 2

```
. . . .
. . Q .
Q Q . .
. . Q Q
```

conflict : 3

Iteration 3

```
. Q . .
. . . Q
Q . . .
. . Q .
```

Temp: 90
Solution found

Code:

```
import random
import math

class SimulatedAnnealing:
    def __init__(self, N, initial_temp=10000, cooling_rate=0.999):
        self.N = N  # Size of the board
        self.initial_temp = initial_temp  # Initial temperature
        self.cooling_rate = cooling_rate  # Cooling rate
        self.state = self.random_state()  # Initial state

    def random_state(self):
        """ Generate a random state with one queen per column. """
        return [random.randint(0, self.N - 1) for _ in range(self.N)]

    def calculate_conflicts(self, state):
```

```
        """ Calculate the number of pairs of queens that are attacking each
other. """
        conflicts = 0
        for i in range(self.N):
            for j in range(i + 1, self.N):
                if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                    conflicts += 1
        return conflicts

    def get_neighbors(self, state):
        """ Generate neighboring states by moving one queen to a random
new position. """
        neighbors = []
        for i in range(self.N):
            new_state = state[:]
            new_pos = random.randint(0, self.N - 1)
            while new_pos == new_state[i]:  # Prevent moving the queen to
the same row
                new_pos = random.randint(0, self.N - 1)
            new_state[i] = new_pos
            neighbors.append(new_state)
        return neighbors

    def acceptance_probability(self, current_conflicts,
neighbor_conflicts, temp):
        """ Calculate the probability of accepting a worse solution. """
        if neighbor_conflicts < current_conflicts:
            return 1.0
        else:
            return math.exp((current_conflicts - neighbor_conflicts) / temp)

    def simulated_annealing(self):
        """ Perform the simulated annealing process. """
        current_state = self.state
        current_conflicts = self.calculate_conflicts(current_state)
        temp = self.initial_temp

        print(f"Initial State: {current_state}")
```

```
        print(f"Initial Conflicts: {current_conflicts}")

        while temp > 1:
            neighbors = self.get_neighbors(current_state)
            next_state = random.choice(neighbors)
            next_conflicts = self.calculate_conflicts(next_state)

            # If the next state is better or with some probability, accept the
worse state
            if self.acceptance_probability(current_conflicts, next_conflicts,
temp) > random.random():
                current_state = next_state
                current_conflicts = next_conflicts

            # Cooling the system down
            temp *= self.cooling_rate

            print(f"Temp: {temp:.4f} | Conflicts: {current_conflicts} |
State: {current_state}")

            # If we have found a solution (0 conflicts), stop the process
            if current_conflicts == 0:
                print("Solution found!")
                break

        return current_state, current_conflicts


# Driver code
if __name___== "__main__":
    try:
        N = int(input("Enter the number of queens (N): "))
        sa = SimulatedAnnealing(N)

        final_state, final_conflicts = sa.simulated_annealing()
        print(f"\nFinal State: {final_state}")
        print(f"Final Conflicts: {final_conflicts}")
```

```
    except ValueError:
        print("Please enter a valid number for N.")
```

Output:

```
Final board:
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . Q . .
. . . Q . . . .

Final heuristic (attacking pairs): 0
✅ Solution found!
```

Program-07

Create a knowledge base using prepositional logic & show that the given query entails the knowledge base or not

Pseudocode

function TT-Entails?(KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic

Symbols ← a list of the proposition symbols in KB & α
return TT-check-ALL (KB, α, Symbol, { })

function TT-check-ALL(KB, α, Symbols, model) returns true or false
if Empty? (Symbols) then
If PL-True? (KB, model) then return PL-True?(α, model)
else return true // when KB is false, always return True
else do
P ← first (symbols)
rest ← Rest (symbols)
return (TT-check-ALL (KB, α, rest, model ∪ {P=true})
and
TT-check-ALL (KB, α, rest, model ∪ {P = false}))

Consider the Question KB that contains the following

$Q \rightarrow P$
$P \rightarrow \neg Q$
$Q \vee R$

i) Construct a truth table that shows the truth value of each senstence in KB & indicate the models in which the KB is true

ii) Does KB Entail R?
iii) Does KB entail $R \rightarrow P$
Does KB entail $Q \rightarrow R$

Creative knowledge based using prepositional knowledge and show the given query entries a knowledge based or not

Algorithm:

i) Truth table for P, Q, R

$2^3 = 8$ combination

P	Q	R	Q → P	P ∼ Q	Q ∨ R	KB
T	T	T	T	F	T	
T	T	F	T	F	T	
T	F	T	F	T	T	True
T	F	F	F	T	F	
F	T	T	F	T	T	
F	T	F	F	T	T	
F	F	T	T	T	T	True
F	F	F	T	T	F	

ii) KB entail R

~~No, KB does not~~

The KB is true in this two Condition

Row			
3	P = True	Q = False	R = True
7	P = False	Q = False	R = True

P	Q	R	KB → R	KB → R → P	Q → R
T	T	T			
T	T	F			
T	F	T	True	True	True
T	F	F			
F	T	T			
F	F	T	True	False	True
F	F	F			

KB → R (yes)

Yes KB entails R as in Both 2 conditions where KB is true Both R having True in Both KB Condition

R → P (No)

In both the KB Condition R → P where R = True, P = True then true; R = True, P = False so False (R → P)

Q → R (yes) (In Both KB Condition Rows)

Q = False R = True

Q = False R = True

Code:

Example - 2

$\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \sim C)$

FF→F

A	B	C	A ∨ C	B ∨ ∼C	KB	α
			F	T	F	F
F	F	F	T	F	F	F
F	F	T	F	T	F	T
F	T	F		T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

```
import itertools

# ----- Helper: evaluate propositional logic sentence -----
def pl_true(expr, model):
    if isinstance(expr, str):
        return model[expr]

    op = expr[0]

    if op == 'not':
        return not pl_true(expr[1], model)
    elif op == 'and':
        return pl_true(expr[1], model) and pl_true(expr[2], model)
    elif op == 'or':
        return pl_true(expr[1], model) or pl_true(expr[2], model)
    elif op == 'implies':
        return (not pl_true(expr[1], model)) or pl_true(expr[2], model)
    else:
        raise ValueError("Unknown operator: " + op)


# ----- Truth table entailment algorithm -----
def tt_entails(KB, query, symbols):
    for values in itertools.product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))

        # If KB is true but query is false -> Not entailed
        if all(pl_true(sentence, model) for sentence in KB):
            if not pl_true(query, model):
                print("Counterexample model:", model)
                return False

    return True


# ----- Define your KB -----
# KB: Q → P,  P → ¬Q,  Q ∨ R
```

```
KB = [
    ('implies', 'Q', 'P'),              # Q → P
    ('implies', 'P', ('not', 'Q')),     # P → ¬Q
    ('or', 'Q', 'R')                    # Q ∨ R
]

symbols = ['P', 'Q', 'R']

# ----- Define queries -----
queries = {
    "R": 'R',
    "R → P": ('implies', 'R', 'P'),
    "Q → R": ('implies', 'Q', 'R')
}

# ----- Run entailment tests -----
for name, q in queries.items():
    result = tt_entails(KB, q, symbols)
    print(f"KB entails {name}: {result}")
```

Output:

```
=== Propositional Logic Entailment Checker ===

Enter number of formulas in KB: 3
Formula 1 (use ~ for NOT, & for AND, | for OR, -> for IMPLIES, <-> for IFF): Q->P
Formula 2 (use ~ for NOT, & for AND, | for OR, -> for IMPLIES, <-> for IFF): P->-Q
Formula 3 (use ~ for NOT, & for AND, | for OR, -> for IMPLIES, <-> for IFF): Q|R

Enter the query formula (alpha): Q

Truth Table:
--------------------------------------------------
P | Q | R | Q->P | P->-Q | Q|R | Q
--------------------------------------------------
True | True | True | True | False | True | True
True | True | False | True | False | True | True
True | False | True | True | False | True | False
True | False | False | True | False | False | False
False | True | True | False | False | True | True
False | True | False | False | False | True | True
False | False | True | True | True | True | False

❌ Counterexample found: {'P': False, 'Q': False, 'R': True}
False | False | False | True | True | False | False
--------------------------------------------------

❌ KB does NOT entail Q
```

Program-08

Implement unification in first order logic

Algorithm:

Lab - 07 Unification 30|10|25

Algorithm

UNIFY (Exp 1 , Exp 2)

Step 1 : check if either expression is variable or constant / are identical

If yes → return empty substitution

Step 2 : If one of them is Variable

→ If the variable occurs inside other expression → Fail (avoid

• Otherwise substitute the Variable with other expression

Step 3 : If both are constants / compound terms then unify the arguments ~~Successfully~~ Recursively

① ~~P(f(x) g(y), y)~~

P(f(x), g(y), y) → Exp1

P(f(g(z)), g(f(a)), f(a)) → Exp2

→ f(x) = f(g(z))

x = g(z)

→ g(y) = g(f(a))

y = f(a)

→ y = f(a)

Θ = { x → g(z) , y → f(a) }

z is free Varible

2) Q(x, f(x))

Q(g(y), y)

x = g(y)

f(x) = y

f(g(y)) = y

∴ y occurs on both side this ~~std~~ cannot be unifiable

3) P(x, g(x))

P(g(y), g(g(z)))

x = g(y)

g(x) = g(g(z))

x = g(z)

g(y) = g(z)

y = z

Θ = { x → g(z) , y → z }

{ x → g(y) , z → y }

It is unifiable

30/10/25

Code:

```
def unify(x, y, theta=None):
    if theta is None:
        theta = {}
    if x == y:
        return theta
    elif is_variable(x):
        return unify_var(x, y, theta)
    elif is_variable(y):
        return unify_var(y, x, theta)
    elif is_compound(x) and is_compound(y):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None
```

```
        for xi, yi in zip(x[1], y[1]):
            theta = unify(xi, yi, theta)
            if theta is None:
                return None
        return theta
    else:
        return None

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif is_variable(x) and x in theta:
        return unify(var, theta[x], theta)
    elif occurs_check(var, x, theta):
        return None
    else:
        theta[var] = x
        return theta

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

def is_compound(x):
    return isinstance(x, tuple) and len(x) == 2

def occurs_check(var, x, theta):
    """Check if var occurs inside x (to avoid infinite recursion)"""
    if var == x:
        return True
    elif is_variable(x) and x in theta:
        return occurs_check(var, theta[x], theta)
    elif is_compound(x):
        return any(occurs_check(var, arg, theta) for arg in x[1])
    else:
        return False
expr1 = ('f', ('x', ('g', ('y',))))
expr2 = ('f', (('g', ('z',)), ('g', ('a',))))

theta = unify(expr1, expr2, {})
print("Substitution θ:", theta)
```

Output:

```
Substitution θ: {'x': ('g', ('z',)), 'y': 'a'}


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

Program-09

Implement Alpha Beta and Minima Maxima Using tic tac toe Game

Code:

```
import math, random

# Initialize the board
board = [" " for _ in range(9)]  # 0-8
positions

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

def is_winner(brd, player):
    win_positions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  #
Columns
        [0, 4, 8], [2, 4, 6]              #
Diagonals
    ]
    return any(all(brd[pos] == player for
pos in line) for line in win_positions)

def is_full(brd):
    return all(cell != " " for cell in brd)

def minimax(brd, depth, is_maximizing):
    if is_winner(brd, "O"):
        return 1
```

```
    elif is_winner(brd, "X"):
        return -1
    elif is_full(brd):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i in range(9):
            if brd[i] == " ":
                brd[i] = "O"
                score = minimax(brd, depth +
1, False)
                brd[i] = " "
                best_score = max(best_score,
score)
        return best_score
    else:
        best_score = math.inf
        for i in range(9):
            if brd[i] == " ":
                brd[i] = "X"
                score = minimax(brd, depth +
1, True)
                brd[i] = " "
                best_score = min(best_score,
score)
        return best_score

def best_move():
    # 70% of the time AI plays a random
(bad) move
    if random.random() < 0.7:
        available = [i for i in range(9) if
board[i] == " "]
        return random.choice(available)
    # 30% of the time AI plays optimally
    best_score = -math.inf
    move = None
```

```
    for i in range(9):
        if board[i] == " ":
            board[i] = "O"
            score = minimax(board, 0, False)
            board[i] = " "
            if score > best_score:
                best_score = score
                move = i
    return move

def play_game():
    print("Welcome to Tic Tac Toe (You
= X, AI = O)")
    print("Hint: You can win easily 😉")
    print_board()

    while True:
        # Human move
        while True:
            try:
                move = int(input("Enter your
move (1-9): ")) - 1
                if 0 <= move <= 8 and
board[move] == " ":
                    board[move] = "X"
                    break
                else:
                    print("Invalid move, try
again.")
            except ValueError:
                print("Please enter a number
from 1 to 9.")

        print_board()

        if is_winner(board, "X"):
            print("🎉 You win! (AI made
mistakes)")
```

```
        break
    if is_full(board):
        print("🤝 It's a draw!")
        break

    # AI move
    print("AI is thinking...")
    ai = best_move()
    board[ai] = "O"
    print_board()

    if is_winner(board, "O"):
        print("🤖 AI wins (rarely)!")
        break
    if is_full(board):
        print("🤝 It's a draw!")
        break

# Run the game
if __name__ == "__main__":
    play_game()
```

Output:

```
Welcome to Tic-Tac-Toe!
Choose the game mode:
1. User vs User
2. User vs System
Enter 1 or 2: 1

Please enter name of player 1: dhanush
Please enter the symbol for dhanush: x
Please enter name of player 2: srujan
Please enter the symbol for srujan: o
It's srujan's turn
srujan, enter a number (1 to 9): 5
|   |   |   |
|   |o  |   |
|   |   |   |
It's dhanush's turn
dhanush, enter a number (1 to 9): 6
|   |   |   |
|   |o  |x  |
|   |   |   |
It's srujan's turn
srujan, enter a number (1 to 9): 7
|   |   |   |
|   |o  |x  |
|o  |   |   |
It's dhanush's turn
dhanush, enter a number (1 to 9): 3
|   |   |x  |
|   |o  |x  |
|o  |   |   |
It's srujan's turn
srujan, enter a number (1 to 9): 9
|   |   |x  |
|   |o  |x  |
|o  |   |o  |
It's dhanush's turn
dhanush, enter a number (1 to 9):
=== Session Ended. Please Run the code again ===
```

Program-10

Create a knowledge base For the first order logic statement and prove the given query using Forward reasoning

Algorithm:

11) Create a knowledge base consisting of FOL Statements & prove the given query using Forward

Man (Marcus)

Pompeian (Marcus)

∀x (pompeian (x) → Roman (x))

∀x (Roman (x) → Loyal (x)]

∀x (Man (x) → Person (x)

∀x (person(x → Mortal (x))

Query Mortal (marcus)

Mortal (marcus)

Man(Marcus)

↑

man (marcus) → Perron (marcus)

↑

Person (marcus) → mortal (marcus)

6/11/25

Code:

```
facts = {
    "Man(Marcus)",
    "Pompeian(Marcus)"
```

```
}

    rules = [
        ("Pompeian(x)", "Roman(x)"),
        ("Roman(x)", "Loyal(x)"),
        ("Man(x)", "Person(x)"),
        ("Person(x)", "Mortal(x)")
    ]

    query = "Mortal(Marcus)"




    def match(statement, fact):
        """
        Match a statement pattern like 'Pompeian(x)' to a fact like
    'Pompeian(Marcus)'.
        Returns the substitution (x -> Marcus) if they match.
        """
        if "(" not in statement or "(" not in fact:
            return None
        pred1, arg1 = statement[:-1].split("(")
        pred2, arg2 = fact[:-1].split("(")
        if pred1.strip() != pred2.strip():
            return None
        if arg1.strip().islower():  # variable like x
            return {arg1.strip(): arg2.strip()}
        elif arg1.strip() == arg2.strip():
            return {}
        else:
            return None



    def substitute(statement, subs):
        for var, val in subs.items():
            statement = statement.replace(var, val)
        return statement
```

```
def forward_chain(facts, rules, query):
    inferred = set()
    while True:
        new_facts = set(facts)
        for antecedent, consequent in rules:
            for fact in facts:
                subs = match(antecedent, fact)
                if subs is not None:
                    new_fact = substitute(consequent, subs)
                    if new_fact not in facts:
                        print(f"Inferred: {new_fact} (from {fact} using rule {antecedent} → {consequent})")
                        new_facts.add(new_fact)
        # If no new facts, stop
        if new_facts == facts:
            break
        facts = new_facts
    return query in facts, facts


result, all_facts = forward_chain(facts, rules, query)

print("\n--- Final Facts ---")
for f in sorted(all_facts):
    print(f)

print("\n--- Result ---")
if result:
    print(f"The query '{query}' is TRUE based on forward reasoning.")
else:
    print(f"+ The query '{query}' cannot be proven from the given knowledge base.")
```

Output:

```
All inferred facts:
Hostile(A)
Criminal(Robert)
Weapon(T1)
Sells(Robert, T1, A)

✅ Robert is a criminal
```

Program-11

Convert the given first order logic statement to conjunctive normal form (CNF)

Algorithm:

10. Convert FOL into CNF

~~Everyone who Loves All animals is loved by Someone~~

~~$\forall x\ (\forall y\ Animal(y) \rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$~~

1) ~~Eliminate Implic~~

Q) $\forall x[\neg Ay \neg(Animal(y) \vee Loves(x,y))] \vee [\exists y\ Loves(y,x)]$

Step1 :- Eliminate Implication ($\rightarrow$)

There are no implication here

step 2 : Move negation inward

$\neg\forall y \neg(Animal(y) \vee Loves(x,y))$ – Remove negate $\neg\forall$ becomes $\exists\neg\exists$

Using Quantifier Negatio Rule

$\neg\forall y\ P(y) = \exists y \neg P(y)$

$\neg\exists y\ P(y) \equiv \forall y \neg P(y)$

$\rightarrow \exists y\ (Animal(y) \vee Loves(x,y))$

Now the whole sensentee becomes

$\forall x\ [\exists y\ (Animal(y) \vee Loves(x,y))] \vee [\exists y\ Loves(y,x)]$

Step 3 : Standardise Variable (unique for each)

$\forall x\ [\exists y\ (Animal(y) \vee Loves(x,y))] \vee [\exists z\ Loves(z,x)]$

Step 4 : Move all Quantifier to front

$\forall x\ \exists y\ \exists z[(Animal(y) \vee Loves(x,y)) \vee Loves(z,x)]$

Step 5 : Skolemization Eliminate Existential Quantifier

Replace $y \rightarrow f(x)$

Replace $z \rightarrow g(x)$ eliminate $\exists$

$\forall x\ [(Animal(f(x)) \vee Loves(x,f(x)) \vee Loves(g(x),x)]$

Step6 : Drop Universal Quantifier $\forall$

$(Animal(f(x)) \vee Loves(x,f(x)) \vee Loves(g(x),x))$

Step 7 : need to distribute $\vee$ over $\wedge$

Already all or present (or) (and)

no need

$(Animal(f(x)) \vee Loves(x,f(x)) \vee Loves(g(x),x))$

Code:

```
import copy

# Utility for deep substitution
def substitute(term, var, replacement):
    """Replace variable var with replacement inside a term."""
    if isinstance(term, str):
        return replacement if term == var else term
    elif isinstance(term, tuple):
        return tuple(substitute(t, var, replacement) for t in term)
    else:
        return term


# 1️⃣ Remove negations using equivalences
def eliminate_negations(expr):
    """Apply ¬∀y¬P(y) ≡ ∃y P(y) and ¬∃y P(y) ≡ ∀y ¬P(y)."""
    if isinstance(expr, tuple):
        op = expr[0]
        if op == 'not':
            sub = expr[1]
            if isinstance(sub, tuple) and sub[0] == 'forall':
                var, inner = sub[1], sub[2]
                return ('exists', var, eliminate_negations(('not', inner)))
            elif isinstance(sub, tuple) and sub[0] == 'exists':
                var, inner = sub[1], sub[2]
                return ('forall', var, eliminate_negations(('not', inner)))
            elif isinstance(sub, tuple) and sub[0] == 'not':
                return eliminate_negations(sub[1])
            else:
                return ('not', eliminate_negations(sub))
        elif op in ['and', 'or']:
            return (op, eliminate_negations(expr[1]), eliminate_negations(expr[2]))
        elif op in ['forall', 'exists']:
            return (op, expr[1], eliminate_negations(expr[2]))
    return expr
```

```
#   Move quantifiers to front (Prenex form)
def move_quantifiers(expr):
    if isinstance(expr, tuple):
        op = expr[0]
        if op in ['and', 'or']:
            left = move_quantifiers(expr[1])
            right = move_quantifiers(expr[2])
            # If quantifiers exist in left or right, move them out
            if isinstance(left, tuple) and left[0] in ['forall', 'exists']:
                return (left[0], left[1], move_quantifiers((op, left[2], right)))
            elif isinstance(right, tuple) and right[0] in ['forall', 'exists']:
                return (right[0], right[1], move_quantifiers((op, left, right[2])))
            else:
                return (op, left, right)
        elif op in ['forall', 'exists']:
            return (op, expr[1], move_quantifiers(expr[2]))
    return expr


#   Skolemization
def skolemize(expr, scope_vars=None):
    """Remove existential quantifiers using Skolem functions."""
    if scope_vars is None:
        scope_vars = []
    if isinstance(expr, tuple):
        op = expr[0]
        if op == 'forall':
            return ('forall', expr[1], skolemize(expr[2], scope_vars + [expr[1]]))
        elif op == 'exists':
            func_name = f"f_{expr[1]}"
            skolem_func = func_name + "(" + ",".join(scope_vars) + ")" if scope_vars else func_name
            return skolemize(substitute(expr[2], expr[1], skolem_func), scope_vars)
        elif op in ['and', 'or']:
            return (op, skolemize(expr[1], scope_vars), skolemize(expr[2], scope_vars))
    return expr
```

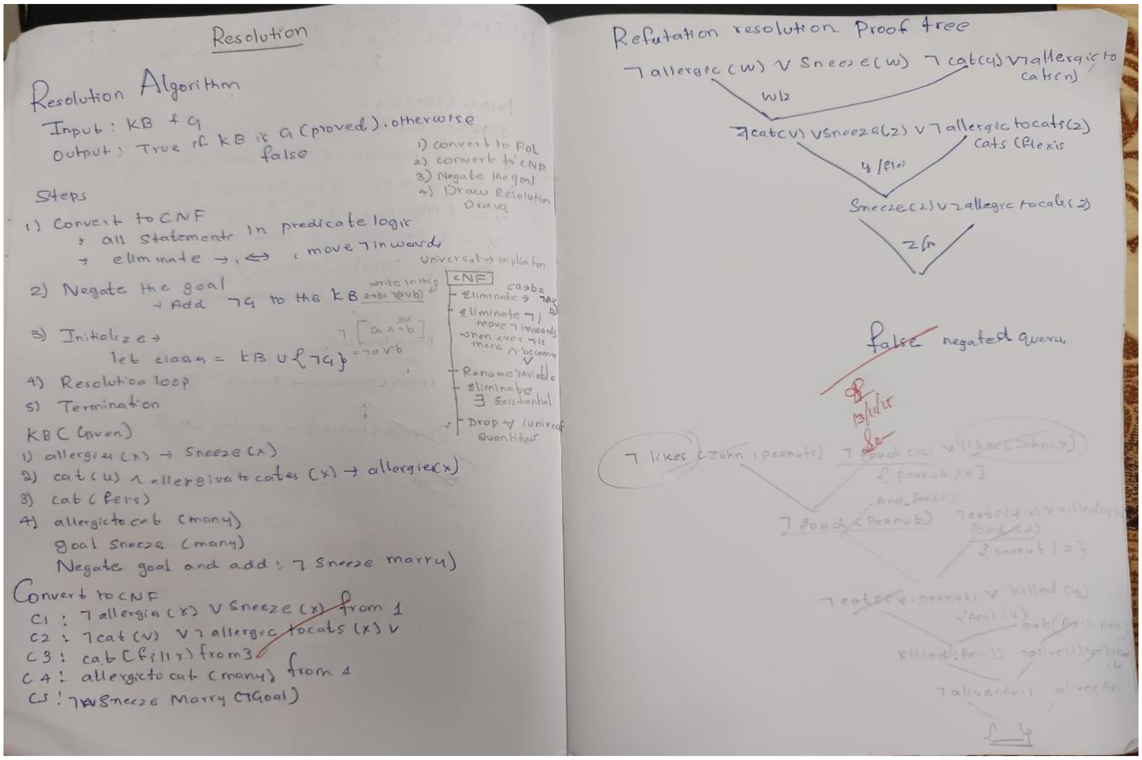
```python
# Drop universal quantifiers
def drop_universal(expr):
    if isinstance(expr, tuple) and expr[0] == 'forall':
        return drop_universal(expr[2])
    elif isinstance(expr, tuple) and expr[0] in ['and', 'or']:
        return (expr[0], drop_universal(expr[1]), drop_universal(expr[2]))
    return expr


# Distribute ∨ over ∧ to get CNF
def distribute_or_over_and(expr):
    if not isinstance(expr, tuple):
        return expr
    op = expr[0]
    if op == 'or':
        a, b = expr[1], expr[2]
        if isinstance(a, tuple) and a[0] == 'and':
            return ('and',
                    distribute_or_over_and(('or', a[1], b)),
                    distribute_or_over_and(('or', a[2], b)))
        elif isinstance(b, tuple) and b[0] == 'and':
            return ('and',
                    distribute_or_over_and(('or', a, b[1])),
                    distribute_or_over_and(('or', a, b[2])))
        else:
            return ('or', distribute_or_over_and(a), distribute_or_over_and(b))
    elif op == 'and':
        return ('and', distribute_or_over_and(expr[1]), distribute_or_over_and(expr[2]))
    else:
        return expr


def to_cnf(expr):
    expr = eliminate_negations(expr)
    expr = move_quantifiers(expr)
    expr = skolemize(expr)
```

```
    expr = drop_universal(expr)
    expr = distribute_or_over_and(expr)
    return expr


expr = ('forall', 'x',
        ('or',
            ('not', ('forall', 'y', ('not', ('or', ('Animal', 'y'), ('Loves', 'x', 'y'))))),
            ('exists', 'y', ('Loves', 'y', 'x'))
        )
)

cnf = to_cnf(expr)
print("Final CNF Structure:\n", cnf)
```

Program 12

Create knowledge piece using prepositional logic and prove the given query using resolution

Algorithm:



Code:

```
import copy
import itertools

# ------------------------------
# Unification
# ------------------------------
def is_variable(x):
    return isinstance(x, str) and x[0].islower()

def unify(x, y, subs=None):
    if subs is None:
        subs = {}
    if x == y:
        return subs
    elif is_variable(x):
        return unify_var(x, y, subs)
    elif is_variable(y):
        return unify_var(y, x, subs)
```

```
elif isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
```

```
        for xi, yi in zip(x, y):
            subs = unify(xi, yi, subs)
            if subs is None:
                return None
        return subs
    else:
        return None

def unify_var(var, x, subs):
    if var in subs:
        return unify(subs[var], x, subs)
    elif x in subs:
        return unify(var, subs[x], subs)
    elif occurs_check(var, x, subs):
        return None
    else:
        subs_copy = subs.copy()
        subs_copy[var] = x
        return subs_copy

def occurs_check(var, x, subs):
    if var == x:
        return True
    elif isinstance(x, list):
        return any(occurs_check(var, xi, subs) for xi in x)
    elif x in subs:
        return occurs_check(var, subs[x], subs)
    return False


# ------------------------------
# Resolution
# ------------------------------
def negate(literal):
    if literal.startswith('~'):
        return literal[1:]
    else:
        return '~' + literal

def substitute(clause, subs):
    new_clause = []
    for literal in clause:
        pred, args = parse_predicate(literal)
        new_args = []
        for a in args:
            if a in subs:
```

```
            new_args.append(subs[a])
         else:
            new_args.append(a)
      new_clause.append(f"{'~' if literal.startswith('~') else ''}{pred}({','.join(new_args)})")
   return new_clause

def parse_predicate(literal):
   neg = literal.startswith('~')
   if neg:
      literal = literal[1:]
   name, args = literal.split('(')
   args = args[:-1].split(',')  # remove ')'
   return name, args

def resolve(ci, cj):
   for di in ci:
      for dj in cj:
         if di.startswith('~') != dj.startswith('~'):  # opposite polarity
            pred_i, args_i = parse_predicate(di)
            pred_j, args_j = parse_predicate(dj)
            if pred_i == pred_j:
               subs = unify(args_i, args_j)
               if subs is not None:
                  new_ci = substitute(ci, subs)
                  new_cj = substitute(cj, subs)
                  new_clause = list(set([x for x in new_ci + new_cj if x != di and x != dj]))
                  return new_clause
   return None

def resolution(kb, query):
   clauses = copy.deepcopy(kb)
   clauses.append([negate(query)])  # negate query for proof by contradiction
   new = set()

   print("\n--- Resolution Steps ---")
   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:
         resolvent = resolve(ci, cj)
         if resolvent == []:
            print("Derived empty clause ⇒ Query proven
            return True
         if resolvent is not None:
            new.add(tuple(sorted(resolvent)))

      new_clauses = [list(x) for x in new if list(x) not in clauses]
```

```
        if not new_clauses:
            print("No new clauses ⇒ Query cannot be proven ❌")
            return False
        for c in new_clauses:
            clauses.append(c)


# -----------------------------
# Example Knowledge Base
# -----------------------------
# KB:
# 1. John likes all kinds of food.
# 2. Apple and vegetable are food.
# 3. Anything anyone eats and not killed is food.
# 4. Anil eats peanuts and still alive.
# 5. Harry eats everything Anil eats.
# 6. Anyone who is alive is not killed.
# 7. Anyone who is not killed is alive.
# Query: John likes peanuts.

kb = [
    ['~Food(x)', 'Likes(John,x)'],              # John likes all food
    ['Food(Apple)'],                            # Apple is food
    ['Food(Vegetable)'],                        # Vegetable is food
    ['~Eats(x,y)', '~Alive(x)', 'Food(y)'],     # Anything eaten by alive person is food
    ['Eats(Anil,Peanuts)'],                     # Anil eats peanuts
    ['Alive(Anil)'],                            # Anil is alive
    ['~Eats(Harry,y)', 'Eats(Anil,y)'],         # Harry eats everything Anil eats
    ['~Alive(x)', '~Killed(x)'],                # Alive(x) -> not Killed(x)
    ['~Killed(x)', 'Alive(x)']                  # not Killed(x) -> Alive(x)
]

query = 'Likes(John,Peanuts)'

print("Converting to CNF and proving using resolution...")
resolution(kb, query)
```

Output:

```
Converting to CNF and proving using resolution...

--- Resolution Steps ---
Derived empty clause ⇒ Query proven ✅
```