

**VISVESVARAYA TECHNOLOGICAL
UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

G M Kusuma(1BM24CS405)

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-2026

**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 **G M Kusuma (1BM24CS405)**, 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

| Sl. No. | Date | Experiment Title | Page No. |
|--------------------|-------------|---|-----------------|
| 1 | 21-08-2025 | Implement Tic –Tac –Toe Game | 4-12 |
| 2 | 21-08-2025 | Implement vacuum cleaner agent | 13-15 |
| 3 | 28-08-2025 | Implement 8 puzzle problems | 16-19 |
| 4 | 11-09-2025 | Implement Iterative deepening search algorithm | 20-22 |
| 5 | 9-10-2025 | Implement Hill Climbing search algorithm to solve N-Queens problem | 23-26 |
| 6 | 9-10-2025 | Simulated Annealing to Solve 8-Queens problem | 27-30 |
| 7 | 16-10-2025 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 31-33 |
| 8 | 30-10-2025 | Implement unification in first order logic | 34-35 |
| 9 | 30-10-2025 | Implement Alpha-Beta Pruning. | 36-39 |
| 10 | 06-11-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 40-43 |
| 11 | 06-11-2025 | First order logic to CNF | 43-49 |
| 12 | 13-11-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 50-53 |

Github Link:

<https://github.com/dhanushk240206/AI>

Program-01

Implement Tic Tac Toe Game

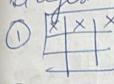
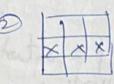
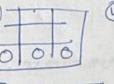
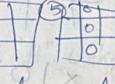
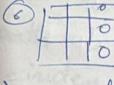
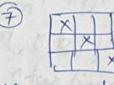
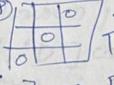
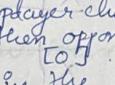
Algorithm:

Lab-1
21/08/25
problem Statement :- understanding the logic of Tic-tac-toe game.

The players can play either both

- Start the program.
- 3x3 board with all 9 positions.

→ The human or Cpu should be understandable as the winning conditions are either row, columns or diagonal there are 8 winning stages.

①  ②  ③  ④ 
⑤  ⑥  ⑦  ⑧ 

→ if player chooses X then opponent O.

→ now place the symbol [X] or [O] in the board.

→ Ask the current player to choose a position

- if the position is empty, place their symbol (X or O). Otherwise, ask again

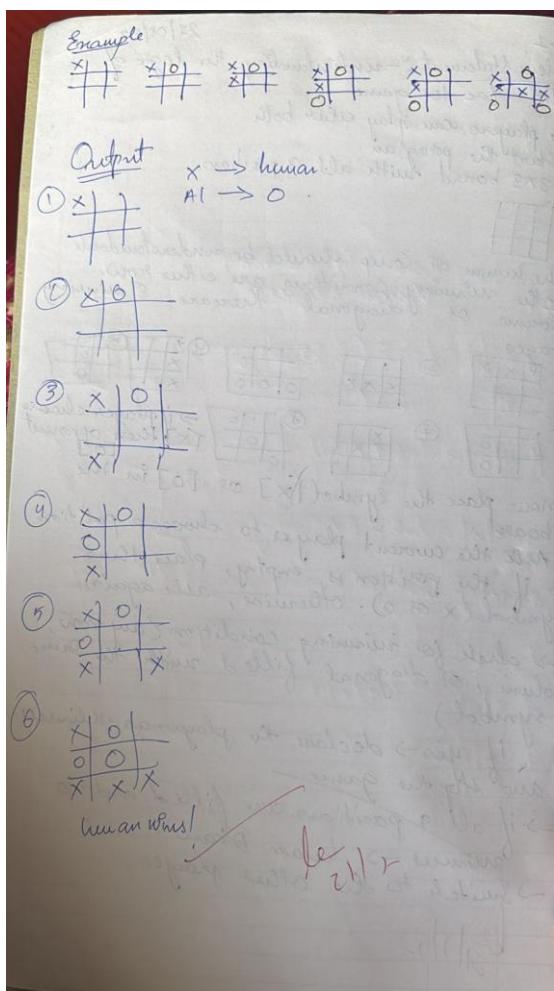
→ check for winning condition (any row, column, or diagonal filled with the same symbol)

if yes → declare the player as winner and stop the game

→ if all 9 positions are filled and no winner → declare Draw.

→ Switch to the other player.

1|1|1
0|1|1



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:

Problem Statement: Vacuum cleaner problem ^{12/2024}
for 4 rooms
→ Start the agent in any one of 4 rooms
(A, B, C, D)
→ check if the current room is Dirty or clean
→ if room is Dirty → perform Suck action
→ if room is clean → move action
→ A to B → B to C → C to D → D to A
→ Repeat steps 2-4 until all rooms are clean.
→ 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 Vacuum Agent(loc, 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]

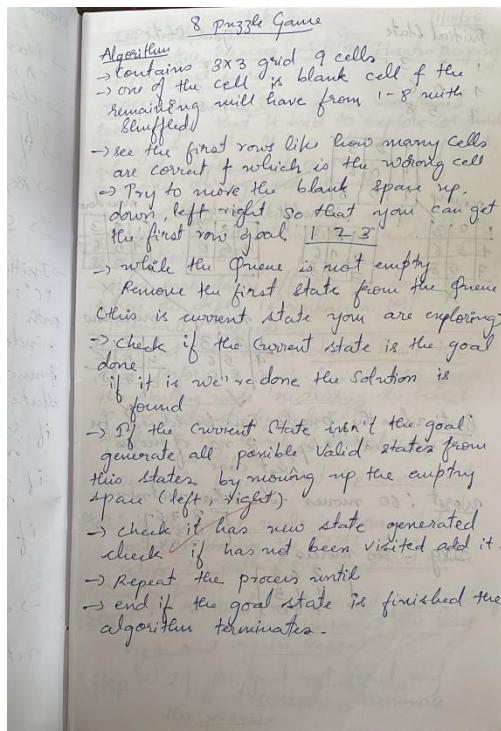
💡 All rooms are clean!

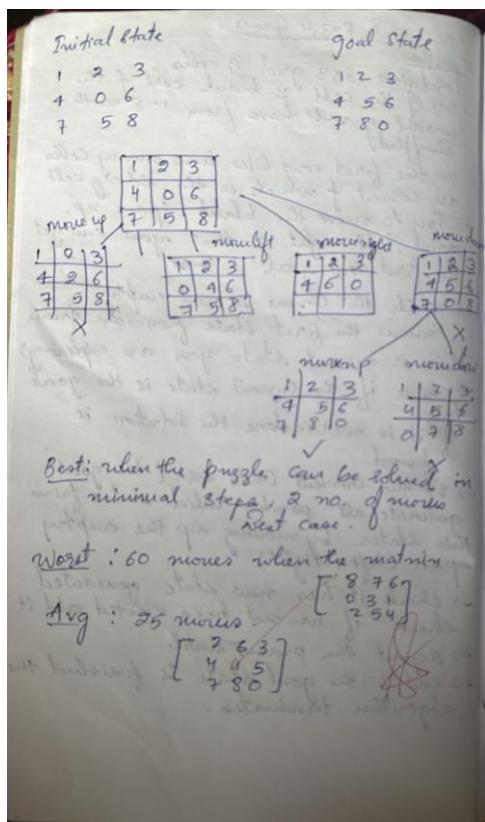
```

Program-03

Implement 8 Puzzle

Algorithm:





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]

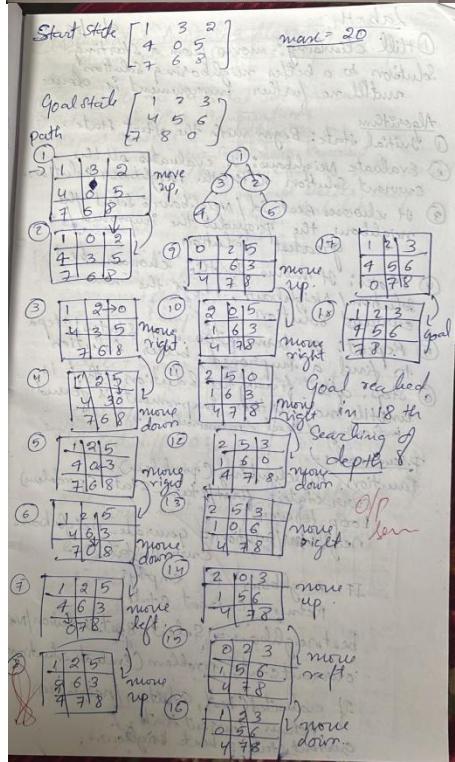
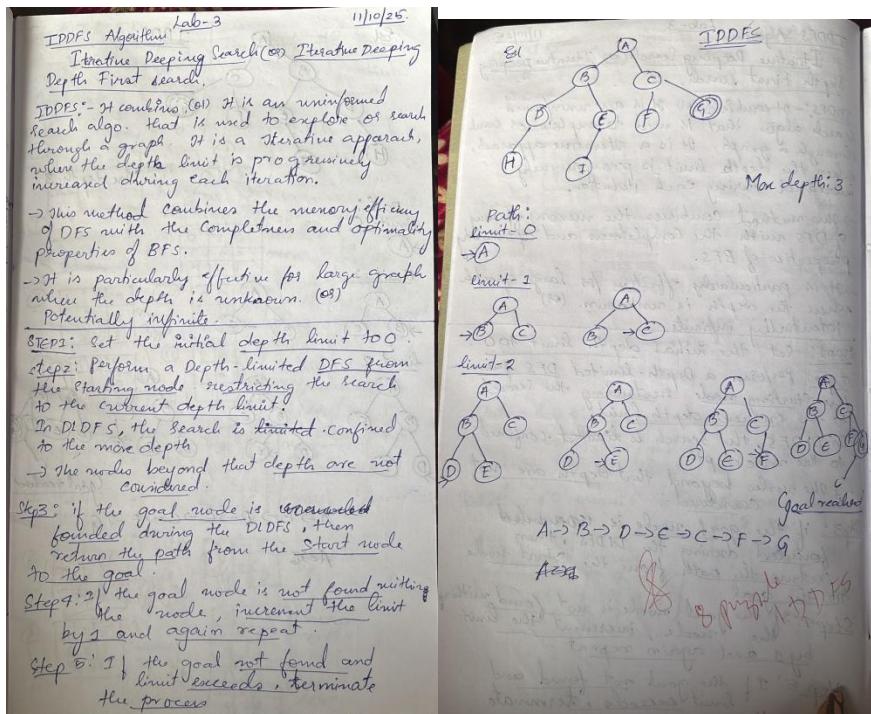
Step 1:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Program-04

Implement IDDFS

Algorithm:



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

Algorithm:

Lab - 4
9/10/25

① Hill climbing :- moves from a starting solution to a better neighboring solution until no further improvement is done.

Algorithm

① Initial state: Begin with the Start State.

② Evaluate Neighbors: It evaluates all the current Solution with the Neighbour Solution.

③ It chooses Best Sol / Neighbour & Select the Neighbour that provides the improved / greatest Solution.

④ Moves: It moves to the chosen, best Solution / Neighbour; it is the new current Solution.

⑤ Repeat: It terminates 2-4 steps/steps to find a improved better Solution.

⑥ Stop: after finding the Solution, that is until no further improvement is found indicating local maximum.

Pseudocode:-

```
Function hillclimbing (problem):
    curr_state = first/start - state (problem)
    loop forever:
        neighbors/solution = generate_neighbors
        current_state
        If neighbors is empty:
            Return current state
        best_neighb = select_best_solution(neighbors)
        current_state, problem (evaluate function)
        If evaluated(best, solution / neighbor)
            Return current state
        current_state = best_neighb.
```

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 .

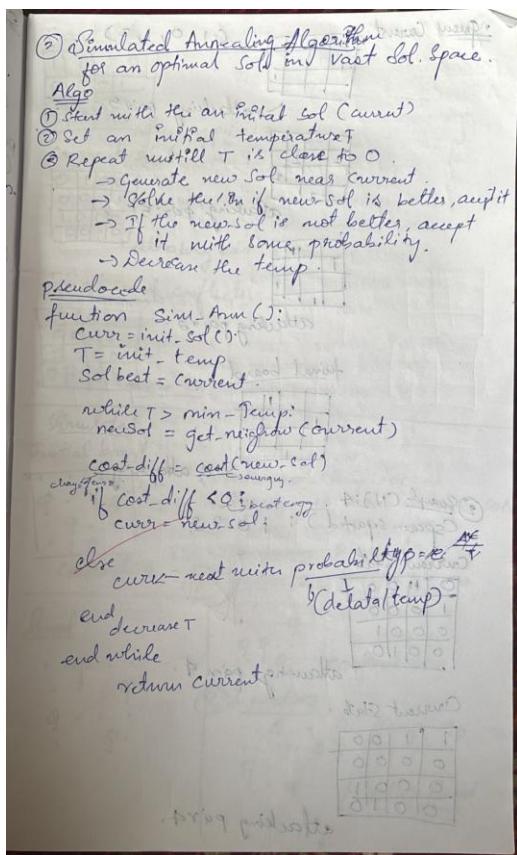
Final heuristic: 1
▲ Local minimum reached (no solution from this start).

```

Program-06

Implement Simulated Annealing

Algorithm:



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 Exception as e:
        print(f'An error occurred: {e}')

```

```
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!
```

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

Algorithm:

propositional logic

Procedure

function $TT\text{-entails}(KB, Query)$
 symbols = all symbols in KB and $Query$
 return $TT\text{-check-All}(KB, Query, symbols, empty model)$

function $TT\text{-check-All}(KB, Query, symbols, model)$
 if symbols is empty then
 if model satisfies KB then
 return model satisfies $Query$
 else
 return true.
 else
 $P \leftarrow \text{first(element of symbols)}$
 rest $\leftarrow \text{Rest(symbols)}$
 return $(TT\text{-check-All}(KB, Query, symbols, model) \vee P = \text{true}) \wedge TT\text{-check-All}(KB, Query, rest, model))$

$\phi \rightarrow \psi \rightarrow P \wedge P \rightarrow \neg \phi \therefore \phi \vee R$

Truth table

(i) KB entail R ?
 (ii) KB entail $R \rightarrow P$?
 (iii) KB entail $R \rightarrow \neg P$?
 (iv) $\neg \neg \phi \rightarrow \neg \phi \rightarrow R \therefore R$

Truth table

| P | ϕ | R | $\phi \rightarrow P$ | $P \sim \phi$ | $\phi \vee R$ | KB |
|-----|--------|-----|----------------------|---------------|---------------|------|
| T | T | T | T | F | T | |
| T | T | F | T | T | T | ✓ |
| T | F | T | T | T | F | |
| T | F | F | T | T | T | |
| F | T | F | F | T | T | |
| F | F | F | F | T | T | ✓ |
| F | F | T | F | T | F | |

b) $\phi \rightarrow P \wedge P \rightarrow \neg \phi$
 and all T.

(i) KB entail R ?
 the KB is true when
 $P \rightarrow T$ $\phi \rightarrow F$ $R \rightarrow T$ $P \rightarrow P \therefore R \rightarrow R$
 $P \rightarrow F$ $\phi \rightarrow F$ $R \rightarrow T$ $F \rightarrow T$

Truth table

| P | ϕ | R | $P \rightarrow R$ | $\neg \phi \wedge R \rightarrow P$ | $\phi \rightarrow R$ |
|-----|--------|-----|-------------------|------------------------------------|----------------------|
| T | T | T | T | ✓ True | T |
| T | T | F | F | ✓ True | T |
| T | F | T | T | ✓ True | T |
| F | T | T | F | False | F |
| F | T | F | F | ✓ True | F |
| F | F | F | F | ✓ True | F |

$PR = \text{true}$.

KB is true
 $R \rightarrow T$

Code:

```

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:

Unification → Lab 7
Algorithm

① check if both exp are identical.
Yes → return empty substitution (no change)

② if one of them is a variable → then substitute it with the other (there should be no circular reference)

③ If both are compound term then unify the arguments recursively.

Q) $\text{Exp}^1 - P(x, f(x), g(y), y)$
sue: $P(f(g(z)), g(f(a)), f(a))$
 $\rightarrow p(x) = f(g(z))$
 $x = g(z)$
 $\rightarrow g(y) = g(f(a))$
 $y = f(a)$
 $\rightarrow y = f(a)$
mgu → $\theta = \{x \rightarrow g(z), y \rightarrow f(a)\}$
 $\exists z$ is a free variable.

Q) $\text{Exp}^2 - g(x, f(x))$
 $g(g(y), y)$
 ~~$x = g(y)$~~
 $f(x) = y$.
 $f(g(y)) = y$
 $\therefore y$ occurs both sides. Not unifiable.

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
        else:
            for i in range(1, len(x)):
                theta = unify(x[i], y[i], theta)
            return theta
    else:
        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:

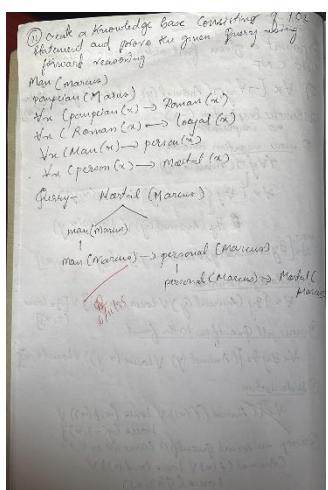
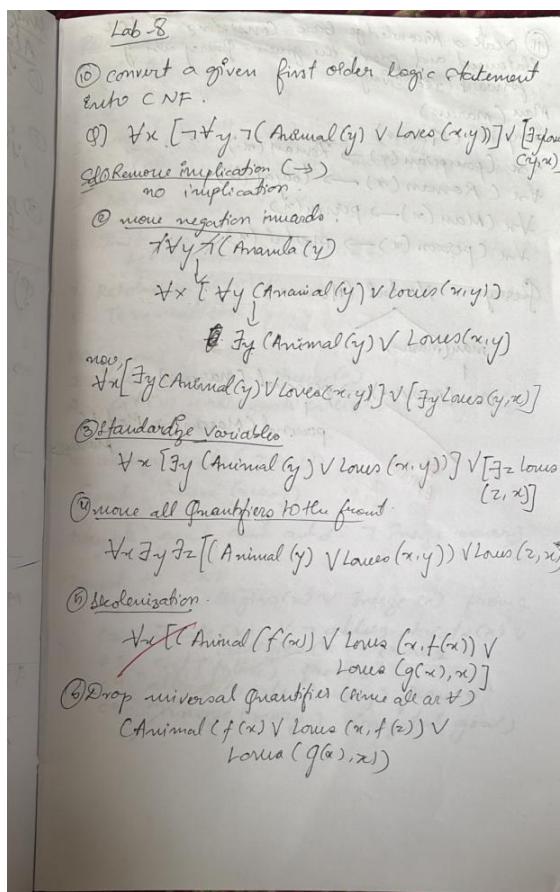
```
Welcome to Tic-Tac-Toe!
Choose the game mode:
1. User vs User
2. User vs System
Enter 1 or 2: 1

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:



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)

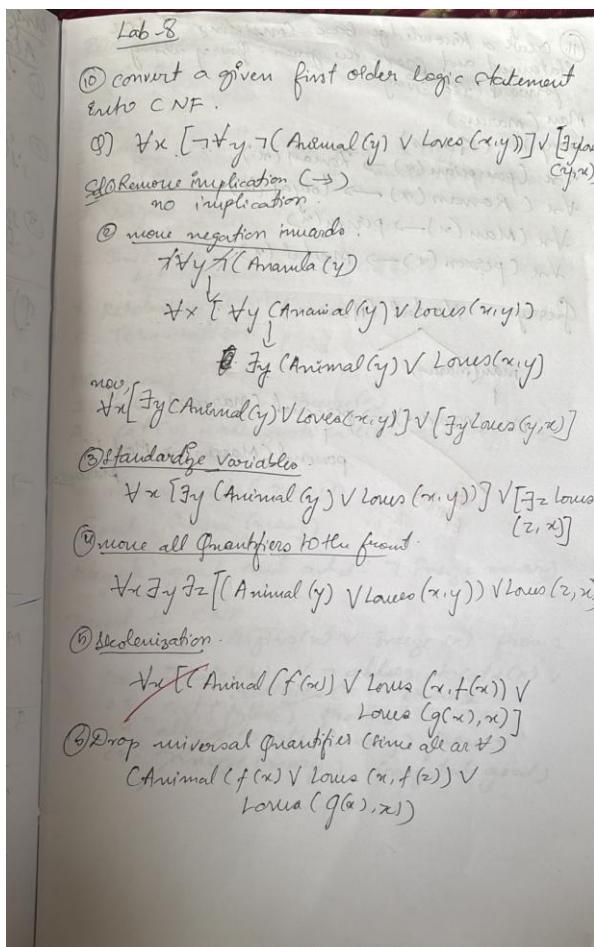
 Robert is a criminal

```

Program-11

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

Algorithm:



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

```

```

# 1 Remove negations using equivalences
def eliminate_negations(expr):
    """Apply  $\neg \forall y P(y) \equiv \exists y \neg P(y)$  and  $\neg \exists y P(y) \equiv \forall y \neg P(y)$ ."""
    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

```

```

# 3 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

```

```
# 3 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
```

```
# 4 Distribute V 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

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 ✓

```