

# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



## LAB REPORT on

### Artificial Intelligence (23CS5PCAIN)

*Submitted by*  
**Chethan K S(1BM23CS074)**

*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 **Chethan K S (1BM23CS074)**, 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

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	21-08-2025	Implement Tic – Tac – Toe Game	4-8
2	21-08-2025	Implement vacuum cleaner agent.	9-11
3	28-08-2025	Solve 8 puzzle problems.	12-14
4	11-09-2025	Implement Iterative deepening search algorithm.	15-17
5	09-10-2025	Implement Hill Climbing Algorithm.	18-20
6	09-10-2025	Write a program to implement Simulated Annealing Algorithm	21-23
7	09-10-2025	Implement A* search algorithm.	24-27
8	16-10-2025	Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.	28-31
9	30-10-2025	Implement unification in first order logic.	32-34
10	30-10-2025	Implement Alpha-Beta Pruning.	35-38
11	06-11-25	Convert a given first order logic statement into Conjunctive Normal Form (CNF).	39-41
12	06-11-25	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	42-47

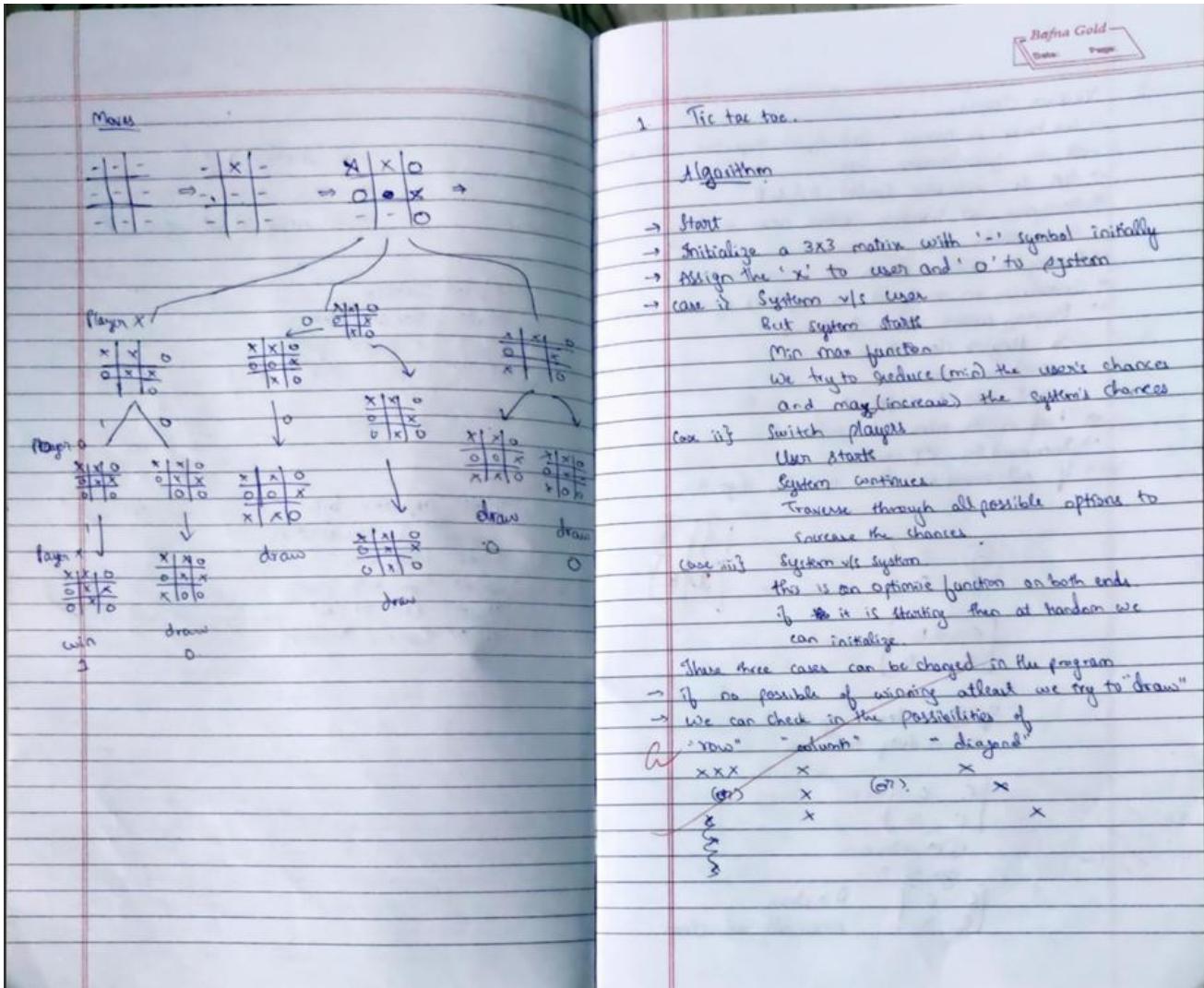
Github Link:

<https://github.com/Chethan-K-S/1BM23CS074-AI-LAB>

## Program 1

Implement Tic - Tac - Toe Game

Algorithm:



Code:

```

import random

# Constants
PLAYER = 'X' # Human
AI = 'O' # System
EMPTY = '-'

# Initialize board
def create_board():

```

```

return [[EMPTY for _ in range(3)] for _ in range(3)]

# Display board
def print_board(board):
    for row in board:
        print(' '.join(row))
    print()

# Check for winner
def check_winner(board):
    lines = []

    # Rows, Columns
    lines.extend(board)
    lines.extend([[board[r][c] for r in range(3)] for c in range(3)])

    # Diagonals
    lines.append([board[i][i] for i in range(3)])
    lines.append([board[i][2 - i] for i in range(3)])

    for line in lines:
        if line.count(line[0]) == 3 and line[0] != EMPTY:
            return line[0]
    return None

# Check if board is full
def is_full(board):
    return all(cell != EMPTY for row in board for cell in row)

# Minimax algorithm
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner == AI:
        return 10 - depth
    elif winner == PLAYER:
        return depth - 10
    elif is_full(board):
        return 0

    if is_maximizing:
        best_score = float('-inf')

```

```

for r in range(3):
    for c in range(3):
        if board[r][c] == EMPTY:
            board[r][c] = AI
            score = minimax(board, depth + 1, False)
            board[r][c] = EMPTY
            best_score = max(score, best_score)
    return best_score

else:
    best_score = float('inf')
    for r in range(3):
        for c in range(3):
            if board[r][c] == EMPTY:
                board[r][c] = PLAYER
                score = minimax(board, depth + 1, True)
                board[r][c] = EMPTY
                best_score = min(score, best_score)
    return best_score

# Best move for AI
def best_move(board):
    best_score = float('-inf')
    move = None
    for r in range(3):
        for c in range(3):
            if board[r][c] == EMPTY:
                board[r][c] = AI
                score = minimax(board, 0, False)
                board[r][c] = EMPTY
                if score > best_score:
                    best_score = score
                    move = (r, c)
    return move

# User move
def user_move(board):
    while True:
        try:
            r, c = map(int, input("Enter row and column (0-2): ").split())
            if board[r][c] == EMPTY:
                return r, c
        except ValueError:
            pass

```

```

else:
    print("Cell already taken.")
except:
    print("Invalid input.")

# System random move (for system vs system)
def random_move(board, symbol):
    empty_cells = [(r, c) for r in range(3) for c in range(3) if board[r][c] == EMPTY]
    return random.choice(empty_cells)

# Game loop
def play_game(mode):
    board = create_board()
    print_board(board)

    current = PLAYER if mode == 'user_vs_system' else AI

    while True:
        if mode == 'user_vs_user':
            print(f"{current}'s turn")
            r, c = user_move(board)

        elif mode == 'user_vs_system':
            if current == PLAYER:
                print("Your turn")
                r, c = user_move(board)
            else:
                print("System's turn")
                r, c = best_move(board)

        elif mode == 'system_vs_system':
            print(f"{current}'s turn")
            r, c = best_move(board) if current == AI else random_move(board, PLAYER)

        board[r][c] = current
        print_board(board)

        winner = check_winner(board)
        if winner:
            print(f"{winner} wins!")
            break

```

```

        elif is_full(board):
            print("It's a draw!")
            break

    current = AI if current == PLAYER else PLAYER

# Choose mode
def main():
    print("Choose mode:")
    print("1. User vs System")
    print("2. System vs System")
    print("3. User vs User")
    choice = input("Enter choice (1-3): ")

    if choice == '1':
        play_game('user_vs_system')
    elif choice == '2':
        play_game('system_vs_system')
    elif choice == '3':
        play_game('user_vs_user')
    else:
        print("Invalid choice.")

```

main()

**Output:**

```

X's turn
Enter row and column (0-2): 1 0
X X 0
X 0 -
-
0 0 -
0 0 0

0's turn
X X 0
X 0 -
0 - -
0 - -

0 wins!

==== Code Execution Successful ====

```

## Program 2

Implement vacuum cleaner agent

Algorithm:

S1(a)25

2. Vacuum cleaner.

- we have 4 rooms , and final objective . to keep all the four rooms clean
- Get 4 variables labeled A,B,C,D
- Assigning at random either clean or dirty to the variable
- Placing the vacuum cleaner at a random room.
- Initializing an empty set to keep track of rooms cleaned
- Display current room & status
  - if dirty → clean it
  - mark as clean
  - and add to empty st
- Check in the order A → B → C → D
- Increment the step counter
- If all rooms are clean end the algorithm.

~~(A) B  
C D~~

~~C → D  
C D~~

~~If (A) → clean go to B  
B → dirty then clean~~

~~A~~

~~[C C]  
[C D]~~

~~(C → clean  
go to D)~~

~~[C C] D → clean  
[C D] now all are clean.~~

**Code:**

```
import random

# Define environment
rooms = {
    'A': random.choice(['Clean', 'Dirty']),
    'B': random.choice(['Clean', 'Dirty']),
    'C': random.choice(['Clean', 'Dirty']),
    'D': random.choice(['Clean', 'Dirty'])
}

# Initial position of the agent
agent_position = random.choice(['A', 'B', 'C', 'D'])

# Display current state
def display_state():
    print(f"Agent is in Room {agent_position}")
    for room, status in rooms.items():
        print(f"Room {room}: {status}")
    print()

# Rule-based agent logic with smart movement
def vacuum_agent():
    global agent_position
    steps = 0

    # Keep track of cleaned rooms
    cleaned_rooms = set()

    while len(cleaned_rooms) < 4:
        display_state()

        # Clean current room if dirty
        if rooms[agent_position] == 'Dirty':
            print(f" ✅ Cleaning Room {agent_position}")
            rooms[agent_position] = 'Clean'
            cleaned_rooms.add(agent_position)
        else:
            print(f" ✅ Room {agent_position} is already clean.")
            cleaned_rooms.add(agent_position)
```

```

# Decide next move only if not all rooms are clean
if len(cleaned_rooms) < 4:
    # Move to the next room that is still dirty
    for next_room in ['A', 'B', 'C', 'D']:
        if next_room not in cleaned_rooms:
            agent_position = next_room
            break

    steps += 1
    print(f"Step {steps} complete.\n")

    print("✿✿✿ All rooms are clean!")
    display_state()

```

vacuum\_agent()

#### Output:

```

Agent is in Room D
Room A: Dirty
Room B: Clean
Room C: Dirty
Room D: Clean

✓ Room D is already clean.
Step 1 complete.

Agent is in Room A
Room A: Dirty
Room B: Clean
Room C: Dirty
Room D: Clean

✓ Cleaning Room A
Step 2 complete.

Agent is in Room B
Room A: Clean
Room B: Clean
Room C: Dirty
Room D: Clean

✓ Room B is already clean.
Step 3 complete.

Agent is in Room C
Room A: Clean
Room B: Clean
Room C: Dirty
Room D: Clean

✓ Cleaning Room C
Step 4 complete.

```

## Program 3

Solve 8 puzzle problems.

Algorithm:

3. 8 puzzle problem.

→ The original state

1	2	3
4	5	6
7	8	0

→ also called as goal state

→ 0 → represents empty Node

→ we take a  $3 \times 3$  grid matrix - index  $\rightarrow i, j$

→ Given problem

0	1	2
1	4	5
2	6	8

→ This to be brought to goal state.

→ Trace the problem  $\rightarrow$  locate where the empty grid "0" is located.

→ 0 located in (1, 1) position

to move down add (1, 0)

to move up add (-1, 0)

to move right add (0, 1)

to move left add (0, -1).

→ We can go through permutations

right move      up move

1	2	3
4	5	6
7	0	8

up move      right move

1	2	3
0	5	6
4	7	8

→

right move      up move      right move      up

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

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

→ Reached goal state.

**Code:**

```
from collections import deque

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

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

def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

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

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

def bfs(start_state):
    visited = set()
    queue = deque()
    queue.append((start_state, []))

    while queue:
        current_state, path = queue.popleft()
        if current_state == goal_state:
            return path + [current_state]

        visited.add(state_to_tuple(current_state))
        x, y = find_blank(current_state)

        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny):
                new_state = [row[:] for row in current_state]
                new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
```

```

        if state_to_tuple(new_state) not in visited:
            queue.append((new_state, path + [current_state]))

    return None

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

```

```
solution_path = bfs(start_state)
```

```

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

```

### Output:

```

PS C:\Users\BMSCECSE\Desktop\Puzzle> & C:/Users/BMSCECSE/AppData/Local/Programs/Python/Python313/python.exe c:/Users/BMSCECSE/Desktop/Puzzle/puzzle.py
Solution found in 3 moves:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
-----
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
-----
PS C:\Users\BMSCECSE\Desktop\Puzzle> []

```

## Program 4

Implement Iterative deepening search algorithm.

**Algorithm:**

n09/25

4. 8 puzzle problem using IDFS

IDDFS:

→ Define the question using a ~~3x3~~ grid matrix.

IDFS algorithm

Search tree diagram:

```

graph TD
    A((A)) -- "d=0" --> B((B))
    A -- "d=0" --> C((C))
    B -- "d=1" --> D((D))
    B -- "d=1" --> E((E))
    C -- "d=1" --> F((F))
    C -- "d=1" --> G((G))
    D -- "d=2" --> H((H))
    D -- "d=2" --> I((I))
    E -- "d=2" --> J((J))
    F -- "d=2" --> K((K))
    G -- "d=2" --> L((L))
    H -- "d=3" --> M((M))
    I -- "d=3" --> N((N))
    J -- "d=3" --> O((O))
    K -- "d=3" --> P((P))
    L -- "d=3" --> Q((Q))
  
```

→ Define depth → levels

→ Start with depth = 0.

- Search only starting node, until you find.
- Increase depth to 1
- Search all nodes reachable in depth 1.
- Increase depth to 2 (go through the nodes of depth 1 again)
- Search all nodes reachable in depth 2.
- Continue the following until you find your goal or reach the max depth.
- In each depth you check the nodes from left to right.

Path:

→ A  
→ A B C  
→ A B D E C F G

8 puzzle → IDDFS

→ Taking H as goal  
→ A  
→ A B C  
→ A B D E C F G  
→ A B D H

→ 8 puzzle → IDFS

Output:

initial: 1 2 3      depth 0 → check initial state  
4 0 6      Depth 1 → [1, 2, 3, 0, 4, 6, 7, 5, 8]  
7 5 8      [1, 2, 3, 4, 5, 6, 7, 8]  
1 2 3      same as original [1, 2, 3, 4, 0, 6, 7, 5, 8]  
4 5 6      Depth 2  
7 0 8      → [1, 2, 3, 4, 5, 6, 7, 8]

goal: 1 2 3  
4 5 6  
7 8 0

Tracing:

1 2 3				
4 0 6				
7 5 8				
Down	1	2	3	
	7	5	8	
	1	2	3	right
	7	5	8	ul
left	1	2	3	
1 2 3	1 2 3	1 2 3	1 2 3	1 2 3
4 5 6	0 4 6	4 6 0	4 2 6	
7 0 8	7 5 8	7 5 8	7 5 8	7 5 8
night	1			
1 2 3				
4 5 6				
7 8 0				

**Code:**

```
from copy import deepcopy

# Define the goal state
GOAL_STATE = [1, 2, 3, 4, 5, 6, 7, 8, 0]

# Generate all valid neighbor states by moving the blank tile
def get_neighbors(state):
    neighbors = []
    index = state.index(0)
    row, col = divmod(index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # up, down, left, right

    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = state[:]
            new_state[index], new_state[new_index] = new_state[new_index], new_state[index]
            neighbors.append(new_state)

    return neighbors

# Depth-Limited Search
def dls(state, depth, visited):
    if state == GOAL_STATE:
        return [state]
    if depth == 0:
        return None
    visited.add(tuple(state))
    for neighbor in get_neighbors(state):
        if tuple(neighbor) not in visited:
            path = dls(neighbor, depth - 1, visited)
            if path:
                return [state] + path
    return None

# Iterative Deepening DFS
def iddfs(start_state, max_depth=20):
    for depth in range(max_depth):
```

```

visited = set()
path = dls(start_state, depth, visited)
if path:
    return path
return None

# Example start state
start = [1, 2, 3, 4, 0, 6, 7, 5, 8]
solution = iddfs(start)

# Print the solution path
if solution:
    print("Solution found in", len(solution) - 1, "moves:")
    for step in solution:
        print(step[:3])
        print(step[3:6])
        print(step[6:])
        print("----")
else:
    print("No solution found within depth limit.")

```

**Output:**

```

PS C:\Users\student\Desktop\Puzzle - iddfs> & "C:\Program Files\Py
Solution found in 2 moves:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
-----
```

## Program 5

Implement Hill Climbing Algorithm.

Algorithm:

### 5. Hill climbing ~~and distributed strategy~~

9/10/25

Algorithm

- Main objective - find the optimal solution of a heuristic function.
- We jump from one node to another node and perform iterations for all the  $n$  nodes. we store it and later compare all the values.
- Pseudo code.

// Define a function

```
int hillclimbing (int n, int val[])
{
    for (int i=0; i<n; i++)
        // iterate the i throughout the nodes
        // give the values in int val[]
        {
            // another for loop
            // compare values using any comparison technique
            return val[i]
        }
}
```

Algorithm

```
function hillclimb()
    current = initial solution()
    current_score = evaluate (current)
    if neighbour_score > current score:
        current = neighbor
        current score = neighbor score
    else break } → while loop (neighbor)
    neighbor = generate - neighbour (current)
    neighbor score = evaluate (neighbor) }
return current.
```

**Code:**

```
import random

# Count number of attacking pairs
def compute_attacks(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

# Generate neighbors by moving one queen in its column
def get_neighbors(state):
    neighbors = []
    for col in range(len(state)):
        for row in range(len(state)):
            if state[col] != row:
                neighbor = state.copy()
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors

# Hill climbing algorithm
def hill_climb():
    current = [random.randint(0, 3) for _ in range(4)]
    current_score = compute_attacks(current)

    while True:
        neighbors = get_neighbors(current)
        next_state = current
        next_score = current_score

        for neighbor in neighbors:
            score = compute_attacks(neighbor)
            if score < next_score:
                next_state = neighbor
                next_score = score

        if next_score == current_score:
```

```

        break # No better neighbor found

    current = next_state
    current_score = next_score

    return current

# Convert solution to matrix
def print_board(state):
    size = len(state)
    board = [['.' for _ in range(size)] for _ in range(size)]
    for col, row in enumerate(state):
        board[row][col] = 'Q'
    for row in board:
        print(' '.join(row))

# Run the algorithm
solution = hill_climb()
print("4 Queens Solution in Matrix Form:")
print_board(solution)

```

### Output:

```

PS C:\Users\student\Desktop\1BM24CS081> & "C:\Program Files\Python312\python.exe" c:/Users/student/Desktop/1BM24CS081/hill
.py
4 Queens Solution in Matrix Form:
. Q . .
. . . Q
Q . . .
. . Q .
PS C:\Users\student\Desktop\1BM24CS081>
fwd-i-search: _

```

## Program 6

Write a program to implement Simulated Annealing Algorithm

Algorithm:

Bafna Gold  
Date: \_\_\_\_\_ Page: \_\_\_\_\_

**6. Simulated annealing**

- Annealing temperature → at the starting iteration the temperature is very high
- Higher the temperature faster the iteration
- As the temperature decreases exploration also decreases
- AnnealingTemp( $n$ )  
Simulated Annealing (initialize):  
item : AnnealingTemp( $n$ )  
    {  
        → while iteration  
             $t = \text{AnnealingTemp} * \text{delta} + n$   
            history.add(heuristic( $n$ )).  
            final = max(heuristic function)  
            return final.  
    }  
}

Algorithm

A. B.

```
function simulatedAnnealing()
    current = initialPoint
    T = initialTemp
    while T > minTemp
        neighbour → generate - neighbor(current)
        ΔE → objective(neighbour) - current
        neighbour
        if ΔE < 0:
            current = neighbour
        else if random(0, 1) < exp(-ΔE/T)
            current = neighbour
    return current
```

**Code:**

```
import random
import math

# Count attacking pairs
def compute_attacks(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

# Generate a neighbor by moving one queen
def get_neighbor(state):
    neighbor = state.copy()
    col = random.randint(0, 3)
    new_row = random.randint(0, 3)
    while neighbor[col] == new_row:
        new_row = random.randint(0, 3)
    neighbor[col] = new_row
    return neighbor

# Simulated Annealing algorithm
def simulated_annealing():
    current = [random.randint(0, 3) for _ in range(4)]
    current_score = compute_attacks(current)
    temperature = 100.0
    cooling_rate = 0.95
    min_temp = 0.01

    while temperature > min_temp and current_score > 0:
        neighbor = get_neighbor(current)
        neighbor_score = compute_attacks(neighbor)
        delta = neighbor_score - current_score

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current = neighbor
            current_score = neighbor_score
```

```

temperature *= cooling_rate

return current

# Print board matrix
def print_board(state):
    board = [['_' for _ in range(4)] for _ in range(4)]
    for col, row in enumerate(state):
        board[row][col] = 'Q'
    for row in board:
        print(' '.join(row))

# Run the algorithm
solution = simulated_annealing()
print("4 Queens Solution (Simulated Annealing):")
print_board(solution)

```

**Output:**

```

4 Queens Solution (Simulated Annealing):
. . Q .
Q . . .
. . . Q
. Q . .

```

## Program 7

Implement A\* search algorithm.

Algorithm:

ct<sup>n</sup> algorithm

```
function A* (start, goal)
    open-set = priority queue with start (start)
    came-from (from → empty map)

    g-score [start-state] = 0
    f-score [start-state] = heuristic (start, start)

    while open-set is not empty,
        current-node is open-set with lowest f-score
        if current = goal-state
            return reconstruct-path (came-from, current)
        remove current.

        take count → path count.
        now done.

    1 2 3
    8 0 5
    u 7 6
    ↓
    1 2 3   1 2 3
    0 8 5   u 5 6
    u 7 6   7 8 0
    ↓           ↑
    1 2 3   1 2 3
    u 8 5   u 8 5
    7 0 6   8 7 6
```

Code:

```
import heapq
```

```

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.moves = moves
        self.previous = previous
        self.size = 3

    def __eq__(self, other):
        return self.board == other.board

    def __hash__(self):
        return hash(self.board)

    def get_neighbors(self):
        neighbors = []
        zero_index = self.board.index(0)
        x, y = divmod(zero_index, self.size)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < self.size and 0 <= new_y < self.size:
                new_zero_index = new_x * self.size + new_y
                new_board = list(self.board)
                new_board[zero_index], new_board[new_zero_index] = new_board[new_zero_index], new_board[zero_index]
                neighbors.append(PuzzleState(tuple(new_board), self.moves + 1, self))
        return neighbors

    def misplaced_tiles(self, goal):
        count = 0
        for i in range(len(self.board)):
            if self.board[i] != 0 and self.board[i] != goal.board[i]:
                count += 1
        return count

    def __lt__(self, other):
        return False

def a_star(start, goal):
    open_set = []

```

```

heapq.heappush(open_set, (start.misplaced_tiles(goal), start))
g_score = {start: 0}
f_score = {start: start.misplaced_tiles(goal)}
closed_set = set()
while open_set:
    current_f, current = heapq.heappop(open_set)
    if current == goal:
        path = []
        while current:
            path.append(current)
            current = current.previous
        path.reverse()
        return path
    closed_set.add(current)
    for neighbor in current.get_neighbors():
        if neighbor in closed_set:
            continue
        tentative_g = g_score[current] + 1
        if neighbor not in g_score or tentative_g < g_score[neighbor]:
            neighbor.previous = current
            g_score[neighbor] = tentative_g
            f = tentative_g + neighbor.misplaced_tiles(goal)
            f_score[neighbor] = f
            heapq.heappush(open_set, (f, neighbor))
return None

```

```

def print_path(path):
    for state in path:
        for i in range(3):
            print(state.board[i*3:(i+1)*3])
    print()

```

```

def get_input_state(prompt):
    print(prompt)
    while True:
        try:
            values = list(map(int, input("Enter 9 numbers (0 for blank) separated by spaces:\n").strip().split()))
            if len(values) != 9 or sorted(values) != list(range(9)):

```

```

        raise ValueError
    return tuple(values)
except ValueError:
    print("Invalid input. Please enter numbers 0 to 8 without duplicates.")

start_board = get_input_state("Enter initial state:")
goal_board = get_input_state("Enter goal state:")

start_state = PuzzleState(start_board)
goal_state = PuzzleState(goal_board)

solution_path = a_star(start_state, goal_state)

if solution_path:
    print(f"Solution found in {len(solution_path)-1} moves:")
    print_path(solution_path)
else:
    print("No solution found.")

```

### Output:

```

✉ Enter initial state:
Enter 9 numbers (0 for blank) separated by spaces: 2 8 3 1 6 4 7 0 5
✉ Enter goal state:
Enter 9 numbers (0 for blank) separated by spaces: 1 2 3 8 0 4 7 6 5
Solution found in 5 moves:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

```

## Program 8

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

Algorithm:

of Propositional logic. 16/10/25

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

KB

$$\begin{array}{l} Q \rightarrow P \\ P \rightarrow \neg Q \\ Q \vee R \end{array}$$

P	Q	R	$Q \rightarrow P$	$P \rightarrow \neg Q$	$Q \vee R$
T	F	T	T	F	T
T	T	F	T	F	T
T	F	T	T	T	T
T	F	F	T	T	F
F	T	T	F	✓T	T
F	T	F	F	✓T	T
F	F	T	T	T	T
F	F	F	T	✓T	F

Does KB entail R?

Does KB entail  $R \rightarrow P$ ?

Does KB entail  $Q \rightarrow R$ ?

Truth table.

P	Q	R	$Q \rightarrow P$	$P \rightarrow \neg Q$	$Q \vee R$
T	F	T	T	F	T
T	T	F	T	F	F
T	F	F	T	T	T
F	T	T	F	✓T	T
F	T	F	F	✓T	T
F	F	T	T	T	T
F	F	F	T	✓T	F

Bafna Gold

→ creation of knowledge base  
→ 3 variables  $[P, Q, R] \rightarrow$  takes values  $(T, F) \times (T, F) \times (T, F)$

# matrix [A]  $\Rightarrow 2 \times 3$  (2 rows, 3 columns)

# permutation  $[P, Q, R]$

if  $A \rightarrow P$  ~~check~~ = T  
if  $P \rightarrow \neg Q$  ~~check~~ = T  
if  $Q \vee R$  ~~check~~ = T  
print (matrix [A])

Now the output value

i	A	C	$Q \rightarrow P$	$P \rightarrow \neg Q$	$Q \vee R$
1	T	T	T	F	F
2	T	F	T	F	P
3	F	T	T	T	T
4	F	F	T	T	F
5	F	T	T	T	T
6	F	F	T	T	F

→ final output.

KB entails R ✓ R exists.  
 $R \rightarrow P$ ? when R is T, P is T ✓  
when R is T, P is F then output is F X

$Q \rightarrow R$ . Q is false A is true so ✓  
 $G \rightarrow V$  ✓

Q1  
Q2/10

Q1 question

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

evaluate (assign formula)

return T or F

generate all possible solution

$$\text{matrix}[x] = \text{permute}[A, S, C] \rightarrow (T, F) \times (T, F) \times (T, F)$$

If  $A \vee C$ ,  $B \vee \neg C$ ,  $KB$  and  $\alpha$  with evaluate.

if  $KB$  true and  $\alpha$  false.  
print ( $KB$  does not entail  $\alpha$ )

if entail true  
print ( $KB$  entails  $\alpha$ )

Output:

A	B	C	$A \vee C$	$B \vee \neg C$	$KB$	$\alpha$
0	0	0	0	1	0	0
0	0	1	1	0	0	0
0	1	0	0	1	0	1
0	1	1	1	1	1	1
1	0	0	1	1	1	1
1	0	1	1	0	0	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

$$|KB| = \alpha \quad 9$$

Code:

```
import itertools
```

```
# Define propositional variables
```

```
variables = ['P', 'Q', 'R']
```

```
# Logical operations
```

```
def implies(a, b):
```

```
    return (not a) or b
```

```
def not_(a):
```

```

return not a

def or_(a, b):
    return a or b

# Evaluate KB sentences
def evaluate_kb_sentences(P, Q, R):
    s1 = implies(Q, P)      #  $Q \rightarrow P$ 
    s2 = implies(P, not_(Q)) #  $P \rightarrow \neg Q$ 
    s3 = or_(Q, R)          #  $Q \vee R$ 
    kb_true = s1 and s2 and s3
    return s1, s2, s3, kb_true

# Generate all models and print truth table
print(f"{'P':<6} {'Q':<6} {'R':<6} {'Q→P':<8} {'P→¬Q':<10} {'Q∨R':<8} {'KB True?':<10}")
print("-" * 54)

kb_true_models = []

for vals in itertools.product([True, False], repeat=3):
    P, Q, R = vals
    s1, s2, s3, kb_true = evaluate_kb_sentences(P, Q, R)
    if kb_true:
        kb_true_models.append({'P': P, 'Q': Q, 'R': R})

print(f"{'str(P)':<6} {'str(Q)':<6} {'str(R)':<6} {'str(s1)':<8} {'str(s2)':<10} {'str(s3)':<8} {'str(kb_true)':<10}")

# Entailment checks
def entails(models, formula_fn):
    return all(formula_fn(model) for model in models)

# Define entailment formulas
def formula_R(model):
    return model['R']

def formula_R_implies_P(model):
    return implies(model['R'], model['P'])

def formula_Q_implies_R(model):
    return implies(model['Q'], model['R'])

```

```
# Print entailment results
print("\nEntailment Results:")
print("KB ⊨ R:", entails(kb_true_models, formula_R))
print("KB ⊨ R → P:", entails(kb_true_models, formula_R_implies_P))
print("KB ⊨ Q → R:", entails(kb_true_models, formula_Q_implies_R))
```

**Output:**

```
PS C:\Users\student> python -u "c:\Users\student\Desktop\IBM24CS074\Propositional logic\propositional logic.py"
P   Q   R   Q→P   P→Q   Q∨R   KB True?
-----
True  True  True  True  False  True  False
True  True  False True  False  True  False
True  False True  True  True  True  True
True  False False True  True  False  False
False True  True  False True  True  False
False True  False False True  True  False
False False True  True  True  True  True
False False False True  True  False  False

Entailment Results:
KB ⊨ R: True
KB ⊨ R → P: False
KB ⊨ Q → R: True
PS C:\Users\student>
```

## Program 9

Implement unification in first order logic.

Algorithm:

30/10/25

9. Unification Algorithm

Algorithm

Steps:

- 1) If  $x$  or  $y$  is a variable or constant, then
  - If  $x$  &  $y$  are same  $\rightarrow$   
 $x=y \rightarrow$  return nil.
  - 2) if  $x$  is a variable
    - \* occurs in  $y \rightarrow$  failure
    - else  $\rightarrow$  ~~unify~~  $\{x/y\}$
  - 3) if  $x$  is variable
    - $y$  occurs in  $x \rightarrow$  failure.
    - else  $\rightarrow$   $\{y/x\}$
  - 4) if  $x \& y$  are compound terms
    - $\rightarrow$  operator or mismatch occurs  $\rightarrow$  failure.
    - else  $\rightarrow$  unify(args( $x$ ), args( $y$ ))
  - 5) if  $x \& y$  are lists.
    - unify heads  $\rightarrow$   $o1 = \text{unify}(x_1, y_1)$
    - if  $o1 = \text{failure} \rightarrow$  return failure.
    - unify tails  $\rightarrow o2 = \text{unify}(o1(x_2), o1(y_2))$ 
      - $\cdot$  if  $o2 = \text{failure} \rightarrow$  return failure
      - else  $\rightarrow$  return compound ( $o2, o1$ )
  - 6) otherwise  $\rightarrow$  return failure.

3 conditions

- 1) Variable vs expression
  - If one side variable  $\rightarrow$  substitute with expression
- 2) Both are compound.
- 3) Lists

Bafna Gold  
Date: \_\_\_\_\_  
Page: \_\_\_\_\_

①  $P(f(x), g(y), z)$   
 $P(b(g(z)), g(f(a)), f(a))$

②  $Q(x, f(x))$   
 $Q(f(g), y)$

③  $P(x, g(x))$   
 $P(g(y), g(g(z)))$

④ Both have 3 arguments.  
 $f(x) \Rightarrow f(g(z))$   
 $\{x \rightarrow g(z)\}$

$g(y) \Rightarrow g(f(a))$   
 $\{y \rightarrow f(a)\}$

$b(a) \Rightarrow y$   
 $\{f(a) \rightarrow y\}$

$\Theta = \{x \rightarrow g(z), y \rightarrow b(a)\}$

$P(f(g(z)), g(f(a)), b(a))$

⑤ → Two arguments.  
 $\{x \rightarrow b(y)\}$   
 $\rightarrow x \rightarrow f(b(y))$

Now unify  $f(b(y))$  and  $y$ .  
 $y$  variable but occurs inside  $f(b(y))$  so not applicable.  
infinite recursion → non unifiable.

⑥ → Two arguments.  
 $\{x \rightarrow g(y)\}$   
 $g(x) \rightarrow g(g(y))$

unify  $g(g(y))$  and  $g(g(z))$   
then  $g(y) \rightarrow g(z)$ .

$\{y \rightarrow z\}$

$\Theta = \{x \rightarrow g(y), y \rightarrow z\}$

or  
 $\Theta = \{x \rightarrow g(z), y \rightarrow z\}$

$P(x, g(x)) \rightarrow$   
 $P(g(z), g(x))$

### Code:

```

def occurs_check(var, expr):
    if var == expr:
        return True
    elif isinstance(expr, tuple):
        return any(occurs_check(var, sub) for sub in expr)
    return False

def unify(x, y, subst={}):
    if subst is None:
        return None
    if x == y:
        return subst
    elif isinstance(x, str) and x.islower(): # x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower(): # y is a variable
        return unify_var(y, x, subst)
    else:
        raise ValueError("Unification failed for non-variable terms")

```

```

        elif isinstance(x, tuple) and isinstance(y, tuple):
            if x[0] != y[0] or len(x) != len(y):
                return None
            for a, b in zip(x[1:], y[1:]):
                subst = unify(apply(subst, a), apply(subst, b), subst)
                if subst is None:
                    return None
                return subst
            else:
                return None

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

def apply(subst, expr):
    if isinstance(expr, str):
        return subst.get(expr, expr)
    elif isinstance(expr, tuple):
        return (expr[0],) + tuple(apply(subst, arg) for arg in expr[1:])
    else:
        return expr

# Represent terms as tuples: ('P', arg1, arg2), ('g', arg), etc.
expr1 = ('P', 'x', ('g', 'x'))
expr2 = ('P', ('g', 'y'), ('g', ('g', 'z')))

result = unify(expr1, expr2)
print("Unification result:", result)

```

### Output:

```

[Running] python -u "c:\Users\BMSCECSE\Desktop\Unify\prg.py"
Unification result: {'x': ('g', 'y'), 'y': 'z'}

[Done] exited with code=0 in 0.138 seconds

```

## Program 10

Implement Alpha-Beta Pruning.

Algorithm:

```

solution Alpha-Beta pruning
10. function alpha-beta (state)
    return maxvalue(state, -∞, +∞)
    v ← maxv(state, -∞, +∞)
    return action in ACTIONS (state) with v
    function MAXV (state, α, β) returns utility value
        if TERMINAL-TEST (state)
            return UTILITY (state)
        v ← -∞
        for each a in ACTIONS (state) do
            v ← MAX (v, MINV (RESULT(s,a), α, β))
            if v ≥ β
                return v
            α ← MAX (α, v)
            return v
        function MINV (state, α, β) returns utility state
            if TERMINAL-TEST (state)
                return utility (state)
            v ← +∞
            for each a in actions(state)
                v ← min(v, maxvalue (s,a), α, β)
            if v ≤ α
                return v
            β ← min(β, v)
            return v.
        Output
        alpha X | X | X | 0
        | 0 | 0 | 0 | X
        | X | X | X | X
        AI at pos 0   AI at pos 6   AI at pos 5
        Me at pos 4   Me at pos 4   Me at pos 1
        A
        X | 0 | 0
        | 0 | X |
        | X | X |
        AI at pos 0   AI at pos 6
        Me at pos 4   Me at pos 4
    
```

Code:

```
import math
```

```
PLAYER = 'X'  
OPPONENT = 'O'
```

```

def check_winner(board):
    win_combos = [
        [0,1,2], [3,4,5], [6,7,8],
        [0,3,6], [1,4,7], [2,5,8],
        [0,4,8], [2,4,6]
    ]
    for combo in win_combos:
        if board[combo[0]] == board[combo[1]] == board[combo[2]] != '':
            return board[combo[0]]
    return None

def is_full(board):
    return '' not in board

def evaluate(board):
    winner = check_winner(board)
    if winner == PLAYER:
        return 10
    elif winner == OPPONENT:
        return -10
    else:
        return 0

def get_children(board, player):
    children = []
    for i in range(9):
        if board[i] == '':
            new_board = board[:]
            new_board[i] = player
            children.append((new_board, i))
    return children

def alpha_beta(board, depth, alpha, beta, maximizing):
    score = evaluate(board)
    if depth == 0 or score in [10, -10] or is_full(board):
        return score

    if maximizing:
        max_eval = -math.inf
        for child, _ in get_children(board, PLAYER):
            eval = alpha_beta(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval

```

```

        return max_eval
    else:
        min_eval = math.inf
        for child, _ in get_children(board, OPPONENT):
            eval = alpha_beta(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
        beta = min(beta, eval)
        if beta <= alpha:
            break
    return min_eval

def find_best_move(board):
    best_val = -math.inf
    best_move = -1
    for child, move in get_children(board, PLAYER):
        move_val = alpha_beta(child, depth=5, alpha=-math.inf, beta=math.inf, maximizing=False)
        if move_val > best_val:
            best_val = move_val
            best_move = move
    return best_move

if __name__ == "__main__":
    board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
    while not is_full(board) and not check_winner(board):
        print("\nCurrent board:")
        print(f"{board[0]} | {board[1]} | {board[2]}")
        print(f"{board[3]} | {board[4]} | {board[5]}")
        print(f"{board[6]} | {board[7]} | {board[8]}")

        move = find_best_move(board)
        board[move] = PLAYER
        print(f"\nAI plays at position {move}")

        if check_winner(board) or is_full(board):
            break

        user_move = int(input("Your move (0-8): "))
        if board[user_move] == ' ':
            board[user_move] = OPPONENT
        else:
            print("Invalid move. Try again.")

        print("\nFinal board:")
        print(f"{board[0]} | {board[1]} | {board[2]}")

```

```
print(f"{board[3]} | {board[4]} | {board[5]}")  
print(f"{board[6]} | {board[7]} | {board[8]}")  
winner = check_winner(board)  
if winner:  
    print(f"Winner: {winner}")  
else:  
    print("It's a draw!")
```

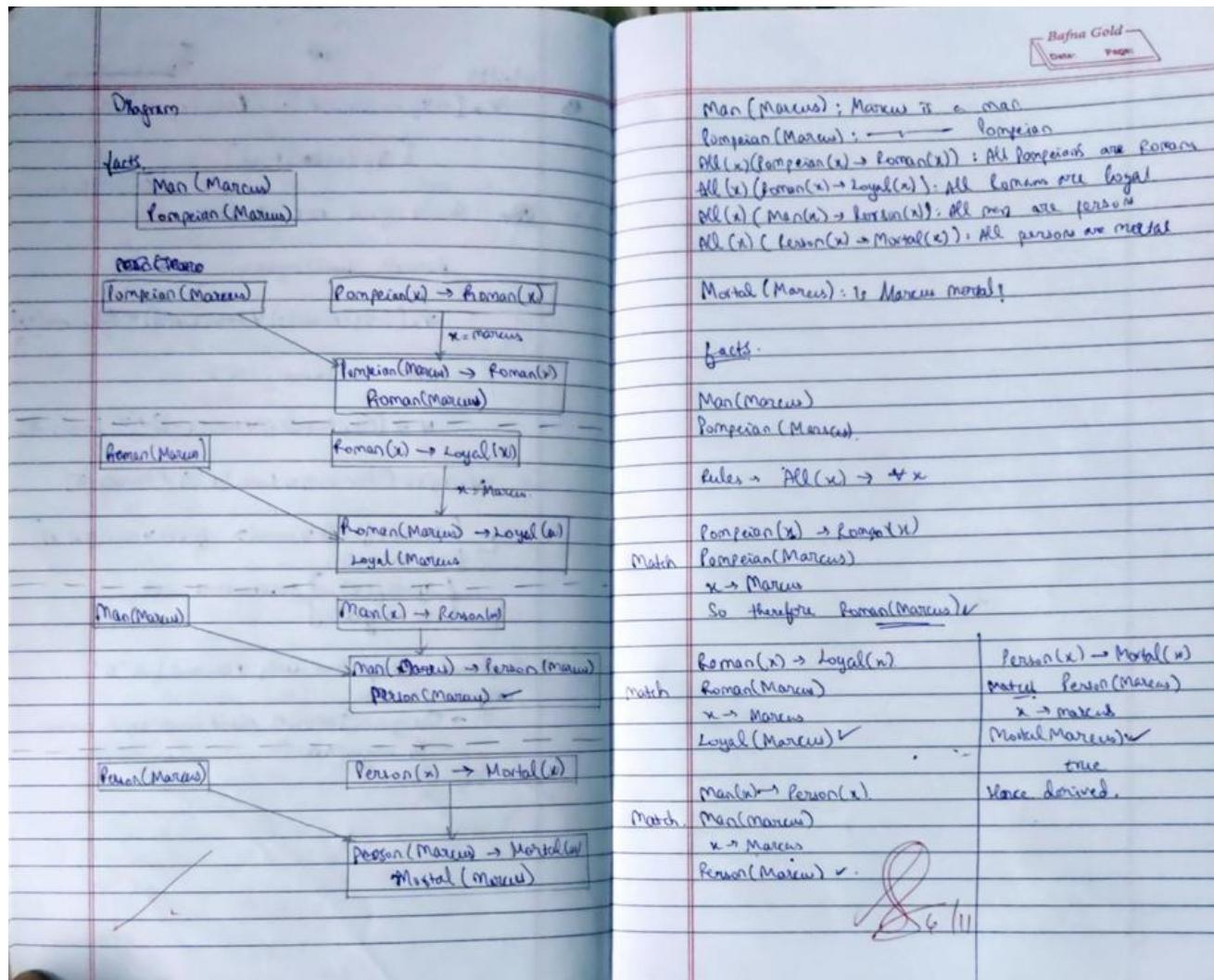
## Output:

```
Current board:  
| |  
| |  
| |  
  
AI plays at position 0  
Your move (0-8): 4  
  
Current board:  
X | |  
| 0 |  
| | |  
  
AI plays at position 1  
Your move (0-8): 2  
  
Current board:  
X | X | 0  
| 0 |  
| | |  
  
AI plays at position 6  
Your move (0-8): 3  
  
Current board:  
X | X | 0  
0 | 0 |  
X | | |  
  
AI plays at position 5  
Your move (0-8): 7  
  
Current board:  
X | X | 0  
0 | 0 | X  
X | 0 | |  
  
AI plays at position 8  
  
Final board:  
X | X | 0  
0 | 0 | X  
X | 0 | X  
It's a draw!  
  
== Code Execution Successful ==
```

## Program 11

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

**Algorithm:**



**Code:**

```
def unify(x, y):
    # A simple unifier (can be replaced by your detailed unify function)
    if x == y:
        return {}
    if isinstance(x, str) and x.islower():
        return {x: y}
```

```

if isinstance(y, str) and y.islower():
    return {y: x}
return "FAILURE"

def apply_substitution(subst, sentence):
    if isinstance(sentence, str):
        return subst.get(sentence, sentence)
    elif isinstance(sentence, dict):
        return {
            'pred': sentence['pred'],
            'args': [apply_substitution(subst, arg) for arg in sentence.get('args', [])]
        }
    return sentence

def sentence_to_str(sentence):
    if isinstance(sentence, str):
        return sentence
    elif isinstance(sentence, dict):
        args_str = ",".join(sentence_to_str(arg) for arg in sentence.get('args', []))
        return f'{sentence["pred"]}({args_str})'
    return str(sentence)

def str_to_sentence(s):
    # Very basic parser assuming format pred(arg1,arg2,...)
    pred_end = s.find("(")
    if pred_end == -1:
        return s
    pred = s[:pred_end]
    args_str = s[pred_end+1:-1]
    args = args_str.split(",") if args_str else []
    return {'pred': pred, 'args': args}

def sentence_in_KB_or_new(sentence, KB, new):
    s_str = sentence_to_str(sentence)
    for rule in KB:
        _, concl = rule
        if sentence_to_str(concl) == s_str:
            return True
    if s_str in new:
        return True
    return False

```

```

def find_substitutions_for_premises(premises, KB):
    # For demo: if no premises, return [{}]
    # (empty substitution)
    if not premises:
        return [{}]
    # Otherwise, just return empty for simplicity
    return [{}]

# Placeholder for the fol_fc_ask function
def fol_fc_ask(KB, alpha):
    """
    A placeholder function for forward chaining inference.
    Replace with your actual implementation.
    """
    print("fol_fc_ask function called.")
    print("Knowledge Base (KB):", KB)
    print("Query (alpha):", alpha)
    # This is where your forward chaining logic would go.
    # For demonstration, let's assume it returns False.
    return False

# Example knowledge base: list of (premises, conclusion)
KB = [
    ([], {'pred': 'prime', 'args': ['11']}), # Fact: prime(11)
    ([{'pred': 'prime', 'args': ['x']}], {'pred': 'odd', 'args': ['x']}), # Rule: prime(x) => odd(x)
]
alpha = {'pred': 'odd', 'args': ['11']} # Query: odd(11)

result = fol_fc_ask(KB, alpha)

print("Result:", result)

```

### Output:

```

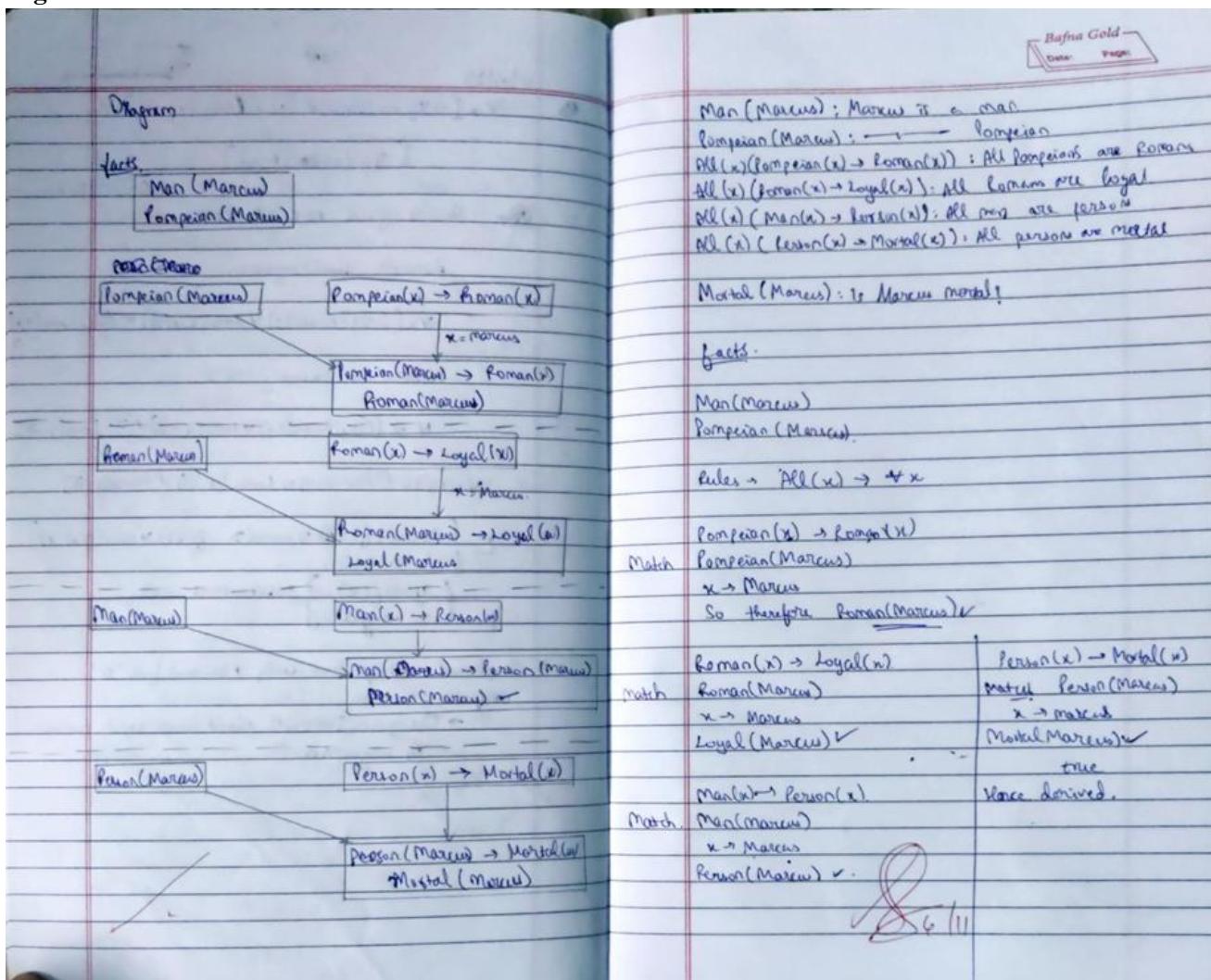
⇒ fol_fc_ask function called.
Knowledge Base (KB): [[[], {"pred": "prime", "args": ["11"]}], ([{"pred": "prime", "args": ["x"]}], {"pred": "odd", "args": ["x"]})]
Query (alpha): {"pred": "odd", "args": ["11"]}
Result: False

```

## Program 12

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

Algorithm:



**Code:**

```
from collections import deque, namedtuple

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

def atom_str(atom):
    pred, args = atom
    return f'{pred}({args})'

# Substitute according to a mapping (var->constant)
def substitute_atom(atom, subs):
    pred, args = atom
    new_args = []
    for a in args:
        if is_variable(a) and a in subs:
            new_args.append(subs[a])
        else:
            new_args.append(a)
    return (pred, tuple(new_args))

# Unification for single terms (simple: variable -> constant only needed here)
def unify_terms(t1, t2, subs):
    # t1, t2 are strings; subs is dict var->const
    t1_val = subs.get(t1, t1) if is_variable(t1) else t1
    t2_val = subs.get(t2, t2) if is_variable(t2) else t2
    if t1_val == t2_val:
        return subs
    # if t1 is variable, bind it
    if is_variable(t1):
        # don't allow binding variable to another variable in this simple scenario; we'll allow var->const only
        subs2 = dict(subs)
        subs2[t1] = t2_val
        return subs2
    if is_variable(t2):
        subs2 = dict(subs)
        subs2[t2] = t1_val
        return subs2
    return None

def unify_atoms(a1, a2):
    # a1,a2 are atoms with same predicate name and same arity
```

```

p1, args1 = a1
p2, args2 = a2
if p1 != p2 or len(args1) != len(args2):
    return None
subs = {}
for x, y in zip(args1, args2):
    subs = unify_terms(x, y, subs)
if subs is None:
    return None
return subs

# Horn rule: head :- body1, body2, ...
Rule = namedtuple("Rule", ["head", "body"]) # head: atom, body: list of atoms (may contain variables)

def forward_chain(rules, facts, query):
    # rules: list of Rule where variables are lowercase strings like 'x'
    # facts: set of ground atoms (constants only)
    # query: atom (ground) to prove
    inferred = set(facts)
    agenda = deque(facts) # facts to be used to try rules
    proof = {} # store how each inferred fact was derived: fact -> (rule, substitutions, premises)

    while agenda:
        fact = agenda.popleft()
        # try every rule: for each rule, try to match a body literal with this fact, then check remaining body literals
        for rule in rules:
            # For each body literal in the rule, attempt to unify it with the current fact
            for i, body_lit in enumerate(rule.body):
                # rename variables in rule to avoid accidental capture (standardize-apart)
                # We'll append a unique suffix for this attempt
                suffix = f"__{id(fact)}_{i}"
                def rename_atom(atom):
                    pred, args = atom
                    new_args = []
                    for a in args:
                        if is_variable(a):
                            new_args.append(a + suffix)
                        else:
                            new_args.append(a)
                    return (pred, tuple(new_args))
                renamed_body = [rename_atom(b) for b in rule.body]
                renamed_head = rename_atom(rule.head)

                target = renamed_body[i]
                subs = unify_atoms(target, fact)

```

```

if subs is None:
    continue
# Now we have a substitution for one body literal; check other body literals
all_ok = True
premises = [fact] # start with matched fact
# For each other literal, try to find a matching ground fact in inferred
for j, other in enumerate(renamed_body):
    if j == i:
        continue
    # we need to find some ground fact in inferred that unifies with 'other' under subs
    matched = False
    for candidate in inferred:
        s2 = unify_atoms(other, candidate)
        if s2 is None:
            continue
        # merge s2 with subs (simple merge; prefer earlier bindings)
        merged = dict(subs)
        conflict = False
        for k,v in s2.items():
            if k in merged and merged[k] != v:
                conflict = True; break
            merged[k] = v
        if conflict:
            continue
        subs = merged
        premises.append(candidate)
        matched = True
        break
    if not matched:
        all_ok = False
        break
if not all_ok:
    continue
# we can fire the rule with substitution 'subs' to produce head
new_head = substitute_atom(renamed_head, subs)
if new_head not in inferred:
    inferred.add(new_head)
    agenda.append(new_head)
    proof[new_head] = (rule, subs, tuple(premises))
    # check query
    if new_head == query:
        return True, inferred, proof
return (query in inferred), inferred, proof

```

# Build KB for Marcus

```

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

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

query = ("Mortal", ("Marcus",))

proved, inferred, proof = forward_chain(rules, facts, query)

print("Forward chaining proof result for query Mortal(Marcus):", proved)
print("\nAll inferred facts:")
for f in sorted(inferred):
    print(" -", atom_str(f))

if proved:
    print("\nProof trace (derived facts and how):")
    # walk backwards from query using the proof dict
    def show_derivation(fact, depth=0):
        indent = " " * depth
        if fact not in proof:
            print(indent + f"{atom_str(fact)} (given)")
            return
        rule_used, subs_used, premises = proof[fact]
        head = rule_used.head
        print(indent + f"{atom_str(fact)} (derived via {atom_str(head)} :- {'.'.join(atom_str(b) for b in rule_used.body)})")
        print(indent + f" substitution: {subs_used}")
        for p in premises:
            show_derivation(p, depth+1)
    show_derivation(query)

```

### **Output:**

```
Forward chaining proof result for query Mortal(Marcus): True
```

```
All inferred facts:
```

- Man(Marcus)
- Mortal(Marcus)
- Person(Marcus)
- Pompeian(Marcus)
- Roman(Marcus)

```
Proof trace (derived facts and how):
```

```
Mortal(Marcus)  (derived via Mortal(x) :- Person(x))
```

```
  substitution: {'x_2573327946496_0': 'Marcus'}
```

```
Person(Marcus)  (derived via Person(x) :- Man(x))
```

```
  substitution: {'x_2573327854976_0': 'Marcus'}
```

```
Man(Marcus)  (given)
```

```
[Done] exited with code=0 in 0.203 seconds
```

