

VISVESVARAYA TECHNOLOGICAL UNIVERSITY
“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Bhoomi Suresh Kota (1BM23CS065)

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
Sep-2024 to Jan-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 **Bhoomi Suresh Kota (1BM23CS065)**, who is a 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	21-8-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4
2	28-8-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	11
3	9-10-2025	Implement A* search algorithm	17
4	9-10-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	21
5	9-10-2025	Simulated Annealing to Solve 8-Queens problem	23
6	16-10-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	26
7	30-10-2025	Implement unification in first order logic	29
8	6-11-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	32
9	13-11-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	36
10	13-11-2025	Implement Alpha-Beta Pruning.	40

Github Link:

<https://github.com/BhoomiSuresh/AI-LAB.git>

Program 1

Implement Tic - Tac - Toe Game
Implement vacuum cleaner agent

Algorithm:

LAB 1	
TIC TAC TOE	
Steps:	
<ul style="list-style-type: none">① Consider a 3×3 matrix② Input can either be 'X' or 'O'③ If the player chooses 'X', system should automatically take 'O' as its input, and vice versa.④ The first moves of the human and system are random.⑤ When human makes the second move, system should check all combinations to check which combination the human can complete and try to block that sequence with by making its move⑥ If none of the sequence can be completed by human, the system should constantly check for if it could complete its sequence and try making a move accordingly	
<i>(Handwritten notes and diagrams related to the steps above)</i>	

Tracing							
LAB 1							
X	O						
X	O						
O	O						
X	X	X	O	X	X	-1	Step 1
X	X	X	O	X	X	-1	Step 2
X	X	X	O	X	X	-1	Step 3
X	X	X	O	X	X	-1	Step 4
X	X	X	O	X	X	-1	Step 5
X	X	X	O	X	X	-1	Step 6
X	X	X	O	X	X	-1	Step 7
X	X	X	O	X	X	-1	Step 8
X	X	X	O	X	X	-1	Step 9
X	X	X	O	X	X	-1	Step 10
X	X	X	O	X	X	-1	Step 11
X	X	X	O	X	X	-1	Step 12
X	X	X	O	X	X	-1	Step 13
X	X	X	O	X	X	-1	Step 14
X	X	X	O	X	X	-1	Step 15
X	X	X	O	X	X	-1	Step 16
X	X	X	O	X	X	-1	Step 17
X	X	X	O	X	X	-1	Step 18
X	X	X	O	X	X	-1	Step 19
X	X	X	O	X	X	-1	Step 20
X	X	X	O	X	X	-1	Step 21
X	X	X	O	X	X	-1	Step 22
X	X	X	O	X	X	-1	Step 23
X	X	X	O	X	X	-1	Step 24
X	X	X	O	X	X	-1	Step 25
X	X	X	O	X	X	-1	Step 26
X	X	X	O	X	X	-1	Step 27
X	X	X	O	X	X	-1	Step 28
X	X	X	O	X	X	-1	Step 29
X	X	X	O	X	X	-1	Step 30
X	X	X	O	X	X	-1	Step 31
X	X	X	O	X	X	-1	Step 32
X	X	X	O	X	X	-1	Step 33
X	X	X	O	X	X	-1	Step 34
X	X	X	O	X	X	-1	Step 35
X	X	X	O	X	X	-1	Step 36
X	X	X	O	X	X	-1	Step 37
X	X	X	O	X	X	-1	Step 38
X	X	X	O	X	X	-1	Step 39
X	X	X	O	X	X	-1	Step 40
X	X	X	O	X	X	-1	Step 41
X	X	X	O	X	X	-1	Step 42
X	X	X	O	X	X	-1	Step 43
X	X	X	O	X	X	-1	Step 44
X	X	X	O	X	X	-1	Step 45
X	X	X	O	X	X	-1	Step 46
X	X	X	O	X	X	-1	Step 47
X	X	X	O	X	X	-1	Step 48
X	X	X	O	X	X	-1	Step 49
X	X	X	O	X	X	-1	Step 50
X	X	X	O	X	X	-1	Step 51
X	X	X	O	X	X	-1	Step 52
X	X	X	O	X	X	-1	Step 53
X	X	X	O	X	X	-1	Step 54
X	X	X	O	X	X	-1	Step 55
X	X	X	O	X	X	-1	Step 56
X	X	X	O	X	X	-1	Step 57
X	X	X	O	X	X	-1	Step 58
X	X	X	O	X	X	-1	Step 59
X	X	X	O	X	X	-1	Step 60
X	X	X	O	X	X	-1	Step 61
X	X	X	O	X	X	-1	Step 62
X	X	X	O	X	X	-1	Step 63
X	X	X	O	X	X	-1	Step 64
X	X	X	O	X	X	-1	Step 65
X	X	X	O	X	X	-1	Step 66
X	X	X	O	X	X	-1	Step 67
X	X	X	O	X	X	-1	Step 68
X	X	X	O	X	X	-1	Step 69
X	X	X	O	X	X	-1	Step 70
X	X	X	O	X	X	-1	Step 71
X	X	X	O	X	X	-1	Step 72
X	X	X	O	X	X	-1	Step 73
X	X	X	O	X	X	-1	Step 74
X	X	X	O	X	X	-1	Step 75
X	X	X	O	X	X	-1	Step 76
X	X	X	O	X	X	-1	Step 77
X	X	X	O	X	X	-1	Step 78
X	X	X	O	X	X	-1	Step 79
X	X	X	O	X	X	-1	Step 80
X	X	X	O	X	X	-1	Step 81
X	X	X	O	X	X	-1	Step 82
X	X	X	O	X	X	-1	Step 83
X	X	X	O	X	X	-1	Step 84
X	X	X	O	X	X	-1	Step 85
X	X	X	O	X	X	-1	Step 86
X	X	X	O	X	X	-1	Step 87
X	X	X	O	X	X	-1	Step 88
X	X	X	O	X	X	-1	Step 89
X	X	X	O	X	X	-1	Step 90
X	X	X	O	X	X	-1	Step 91
X	X	X	O	X	X	-1	Step 92
X	X	X	O	X	X	-1	Step 93
X	X	X	O	X	X	-1	Step 94
X	X	X	O	X	X	-1	Step 95
X	X	X	O	X	X	-1	Step 96
X	X	X	O	X	X	-1	Step 97
X	X	X	O	X	X	-1	Step 98
X	X	X	O	X	X	-1	Step 99
X	X	X	O	X	X	-1	Step 100

Vacuum Cleaner					
Working:	(27B) problem				
① The vacuum cleaner is placed at a point.	It moves in a straight line indefinitely until it encounters an obstacle.				
② It checks right and left directions for dirt. If dirt is detected, moves to that direction and cleans it.	→ Loop				
③ Upon collision, it moves few steps back, and again the directions for dirt.	→ Loop				
④ Direction of rotation is clockwise.	→ Loop				
⑤ This process goes on in a loop.	→ Loop				
<table border="1"> <tr> <td>0</td><td>0</td> </tr> <tr> <td>0</td><td>0</td> </tr> </table>	0	0	0	0	Starts at position 0. Checks left direction If dirt is detected moves left. Status changes to 1 (cleaned)
0	0				
0	0				
<table border="1"> <tr> <td>1</td><td>0</td> </tr> <tr> <td>0</td><td>0</td> </tr> </table>	1	0	0	0	Checks left direction for dirt. Dirt is detected & status changed to 1
1	0				
0	0				
<table border="1"> <tr> <td>1</td><td>1</td> </tr> <tr> <td>0</td><td>0</td> </tr> </table>	1	1	0	0	While moving in clockwise direction, it goes back to initial position, moves through different position for 1 more round. If status of all positions are 1. Then it stops
1	1				
0	0				

Code:

Tic Tac Toe:

```
def computer_move():
    best_move = minimax_recurse(game_board, active_player, 0)
    print("The best move is ", best_move)
    make_move(game_board, best_move, active_player)

    print("COMPUTER MOVE DONE")
```

```
def minimax_recurse(game_board, player, depth):
```

```
    winner = is_winner(game_board)
    if winner == active_player:
        return 1
    elif winner is not None and winner != active_player:
        return -1
    elif len(get_move_list(game_board)) == 0:
```

```

return 0

if player == player1:
    other_player = player2
else:
    other_player = player1

if player == active_player:
    alpha = -1
else:
    alpha = 1

movelist = get_move_list(game_board)
best_move = None

for move in movelist:
    board2 = [list(row) for row in game_board] # Create a copy of the board

    make_move(board2, move, player)

    subalpha = minimax_recurse(board2, other_player, depth + 1)

    if player == active_player:
        if depth == 0 and alpha <= subalpha:
            best_move = move
            alpha = max(alpha, subalpha)
        if alpha == 1: # Alpha-beta pruning
            if depth == 0:
                return best_move
            return alpha
    else:
        alpha = min(alpha, subalpha)
        if alpha == -1: # Alpha-beta pruning
            return alpha

    if depth == 0:
        return best_move
    return alpha


# BOARD FUNCTIONS
game_board = [['1','2','3'],['4','5','6'],['7','8','9']] # Changed to strings to avoid confusion with available moves
def print_board(board) :

    for row in board :

```

```

print(row)

def make_move(game_board, player_move, active_player):

    x = 0
    y = 0
    try:
        player_move = int(player_move)
    except ValueError:
        print("Invalid input. Please enter a number.")
        return game_board

    if player_move == 1 :
        x = 0
        y = 0
    elif player_move == 2 :
        x = 0
        y = 1
    elif player_move == 3 :
        x = 0
        y = 2
    elif player_move == 4 :
        x = 1
        y = 0
    elif player_move == 5 :
        x = 1
        y = 1
    elif player_move == 6 :
        x = 1
        y = 2
    elif player_move == 7 :
        x = 2
        y = 0
    elif player_move == 8 :
        x = 2
        y = 1
    elif player_move == 9 :
        x = 2
        y = 2
    else :
        print ("value is out of range")
        return game_board

    if game_board[x][y] == "O" or game_board[x][y] == "X" :
        print("move not available")
        return game_board

```

```

game_board[x][y] = str(active_player)
return game_board

def is_winner(board):
    for i in range (0,3) :
        if board[i][0] == player1 and board[i][1] == player1 and board[i][2] == player1 :
            return player1

        if board[i][0] == player2 and board[i][1] == player2 and board[i][2] == player2 :
            return player2

    # checking for columns
    for i in range(0,3):
        if board[0][i] == player1 and board[1][i] == player1 and board[2][i] == player1:
            return player1
        if board[0][i] == player2 and board[1][i] == player2 and board[2][i] == player2:
            return player2

    # checking for diagonals
    if board[0][0] == player1 and board[1][1] == player1 and board[2][2] == player1 :
        return player1
    if board[0][0] == player2 and board[1][1] == player2 and board[2][2] == player2 :
        return player2

    if board[2][0] == player1 and board[1][1] == player1 and board[0][2] == player1 :
        return player1
    if board[2][0] == player2 and board[1][1] == player2 and board[0][2] == player2 :
        return player2

    return None

def get_move_list (game_board) :

    move = []

    for row in game_board :
        for i in row :
            if i.isdigit():
                move.append(int(i))
    return move

# Main Loop
player1 = "X"
player2 = "O"
print_board(game_board)
while True :

```

```

active_player = player1
# this is for player move
print(get_move_list(game_board))
player_move = input("Please insert your move >>> ")
game_board = make_move(game_board,player_move,active_player)
print_board(game_board)

if is_winner(game_board) == player1 :
    print("Player1 is the winner")
    break
if is_winner(game_board) == player2 :
    print("Player2 is the winner")
    break
if len(get_move_list(game_board)) == 0:
    print("It's a tie!")
    break

print(get_move_list(game_board))
# computer time
active_player = player2
computer_move()
print_board(game_board)

if is_winner(game_board) == player1 :
    print("Player1 is the winner")
    break
if is_winner(game_board) == player2 :
    print("Player2 is the winner")
    break
if len(get_move_list(game_board)) == 0:
    print("It's a tie!")
    break

```

Vacuum Cleaner:

```

import random

def vacuum_cleaner_agent(location, status_A, status_B):
    """
    Simulates a simple vacuum cleaner agent in a two-location environment (A and B).
    Args:
        location (str): The current location of the vacuum ('A' or 'B').
        status_A (str): The cleanliness status of location A ('Dirty' or 'Clean').
        status_B (str): The cleanliness status of location B ('Dirty' or 'Clean').
    Returns:
        tuple: A tuple containing the updated location and statuses after the agent's action.
    """

```

```

print(f"Vacuum cleaner is at {location}.")

if location == 'A':
    if status_A == 'Dirty':
        print("Location A is Dirty. Sucking dirt...")
        status_A = 'Clean'
        print("Location A is now Clean.")
        # Move to B after cleaning A
        print("Moving to Location B.")
        return 'B', status_A, status_B
    else:
        print("Location A is Clean. Moving to Location B.")
        return 'B', status_A, status_B
elif location == 'B':
    if status_B == 'Dirty':
        print("Location B is Dirty. Sucking dirt...")
        status_B = 'Clean'
        print("Location B is now Clean.")
        # Move to A after cleaning B
        print("Moving to Location A.")
        return 'A', status_A, status_B
    else:
        print("Location B is Clean. Moving to Location A.")
        return 'A', status_A, status_B

# Initializing the environment
initial_location = random.choice(['A', 'B'])
initial_status_A = random.choice(['Dirty', 'Clean'])
initial_status_B = random.choice(['Dirty', 'Clean'])

print(f"Initial state: Vacuum at {initial_location}, A is {initial_status_A}, B is {initial_status_B}\n")

current_location, current_status_A, current_status_B = initial_location, initial_status_A,
initial_status_B

# Running the simulation until both locations are clean
while current_status_A == 'Dirty' or current_status_B == 'Dirty':
    current_location, current_status_A, current_status_B = vacuum_cleaner_agent(
        current_location, current_status_A, current_status_B
    )
    print(f"Current state: Vacuum at {current_location}, A is {current_status_A}, B is {current_status_B}\n")

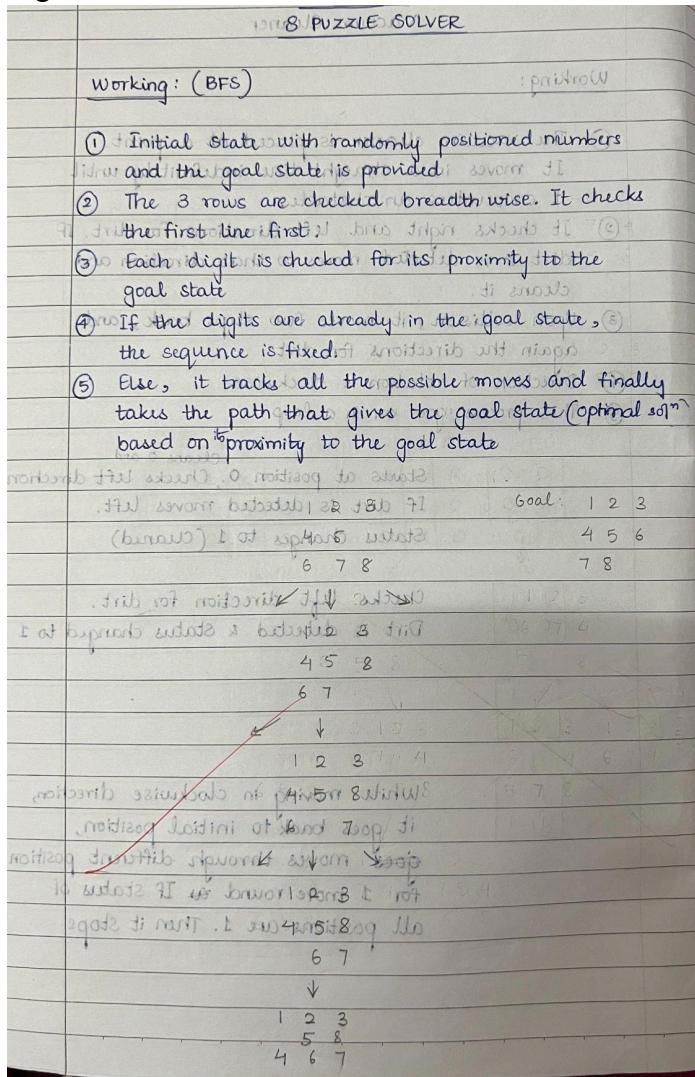
print("Both locations are clean. Vacuum cleaner has finished its task.")

```

Program 2

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

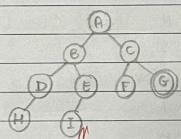
Algorithm:



IDDFS Graph Algorithm:

- 1: Initialize the start and goal node
- 2: Starting from start node till goal node:
 - Start traversing using BFS
 - From the first node, traverse to ~~the~~ its child nodes in the next depth level (DFS)
 - Further, traverse to its child nodes
 - Once we reach the end of the child nodes, move to the next node in the previous breadth.
- 3: If the goal node is found, return the path
Else, if max_depth is reached, return goal not found.

Eq:



Sol: Start from A (BFS[A])

DFS(A) gives B, C

DFS[B] gives D, E

DFS[C] gives F, G

G is the goal state

~~C~~ Path: A → B → D → E → C → F → G

A → B → D → E → C → F → G → H → I

For I :

A → B → D → H → E → I → C → F → G

Output:

M	W	T	F	S	S
				YOUVA	
Page No.:	Date:				

Tile: 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

↓ 4 10 9 8 11 5 6 7 0 8 → 1 2 3

Code:

BFS

```
# Import necessary libraries
from collections import deque

# Define the dimensions of the puzzle
N = 3

# Class to represent the state of the puzzle
class PuzzleState:
    def __init__(self, board, x, y, depth):
        self.board = board
        self.x = x
        self.y = y
        self.depth = depth

    # Possible moves: Left, Right, Up, Down
    row = [0, 0, -1, 1]
    col = [-1, 1, 0, 0]

    # Function to check if the current state is the goal state
    def is_goal_state(board):
        goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
        return board == goal

    # Function to check if a move is valid
    def is_valid(x, y):
        return 0 <= x < N and 0 <= y < N

    # Function to print the puzzle board
    def print_board(board):
        for row in board:
            print(''.join(map(str, row)))
        print('-----')

    # BFS function to solve the 8-puzzle problem
    def solve_puzzle_bfs(start, x, y):
        q = deque()
        visited = set()

        # Enqueue initial state
        q.append(PuzzleState(start, x, y, 0))
        visited.add(tuple(map(tuple, start)))

        while q:
```

```

curr = q.popleft()

# Print the current board state
print(f'Depth: {curr.depth}')
print_board(curr.board)

# Check if goal state is reached
if is_goal_state(curr.board):
    print(f'Goal state reached at depth {curr.depth}')
    return

# Explore all possible moves
for i in range(4):
    new_x = curr.x + row[i]
    new_y = curr.y + col[i]

    if is_valid(new_x, new_y):
        new_board = [row[:] for row in curr.board]
        new_board[curr.x][curr.y], new_board[new_x][new_y] = new_board[new_x][new_y],
        new_board[curr.x][curr.y]

        # If this state has not been visited before, push to queue
        if tuple(map(tuple, new_board)) not in visited:
            visited.add(tuple(map(tuple, new_board)))
            q.append(PuzzleState(new_board, new_x, new_y, curr.depth + 1))

print('No solution found (BFS Brute Force reached depth limit)')

# Driver Code
if __name__ == '__main__':
    start = [[1, 2, 3], [4, 0, 5], [6, 7, 8]] # Initial state
    x, y = 1, 1

    print('Initial State:')
    print_board(start)

    solve_puzzle_bfs(start, x, y)

```

IDDFS:

```

from collections import deque

# Goal state
GOAL = (1, 2, 3,
        4, 5, 6,
        7, 8, 0)

```

```

# Moves: up, down, left, right
MOVES = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

def is_valid_move(pos, move):
    """Check if blank can move in given direction."""
    if move == 'U' and pos < 3: return False
    if move == 'D' and pos > 5: return False
    if move == 'L' and pos % 3 == 0: return False
    if move == 'R' and pos % 3 == 2: return False
    return True

def neighbors(state):
    """Generate valid neighbor states from current state."""
    new_states = []
    pos = state.index(0) # blank position
    for move, offset in MOVES.items():
        if is_valid_move(pos, move):
            new_pos = pos + offset
            new_state = list(state)
            new_state[pos], new_state[new_pos] = new_state[new_pos], new_state[pos]
            new_states.append((tuple(new_state), move))
    return new_states

def dls(state, depth, visited, path):
    """Depth-limited DFS."""
    if state == GOAL:
        return path
    if depth == 0:
        return None
    visited.add(state)

    for neighbor, move in neighbors(state):
        if neighbor not in visited:
            result = dls(neighbor, depth - 1, visited, path + [move])
            if result is not None:
                return result
    return None

def iddfs(start, max_depth=50):
    """Iterative Deepening DFS."""
    for depth in range(max_depth):

```

```

visited = set()
path = dls(start, depth, visited, [])
if path is not None:
    return path
return None

# --- MAIN PROGRAM ---
if __name__ == "__main__":
    print("Enter the start state of the 8-puzzle (use 0 for blank):")
    nums = []
    for i in range(9):
        nums.append(int(input(f"Tile {i+1}: ")))
    start = tuple(nums)

solution = iddfs(start, max_depth=30)
if solution:
    print(f"\nSolution found in {len(solution)} moves:")
    print(" -> ".join(solution))
else:
    print("No solution found within depth limit.")

```

Program 3

Implement A* search algorithm

Algorithm:

A* Algorithm	
Algorithm:	: problem with 1 0 2 0 0
1: initial puzzle state	0 0 0 0
2: priority queue min-heap ordered by $f(n) = g(n) + h(n)$	g(n) ← cost from start to curr-state h(n) ← heuristic (Manhattan distance)
3: if min = f(n) $f(n)$	pilou and pilou with 3 (3) x = min-heap.pop() 0 2 0 0
4: if $x = \text{goal state}$	0 0 0 0 return path 0 0 2 0
else	update position
5: for state in not visited:	calculate $g(n) + h(n)$ $f(n) = g(n) + h(n)$ min-heap.push($f(n)$)
Output	
Initial: [1, 2, 3, 4, 5, 6, 0, 7, 8]	
Move: None	1 2 3 4 5 6 7 8 Solve
Move: R	1 2 3 4 5 6 7 8
Move: R	1 2 3 4 5 6 7 8

Code:

```
import heapq
from termcolor import colored

# Class to represent the state of the 8-puzzle
class PuzzleState:
    def __init__(self, board, parent, move, depth, cost):
        self.board = board # The puzzle board configuration
        self.parent = parent # Parent state
        self.move = move # Move to reach this state
        self.depth = depth # Depth in the search tree
        self.cost = cost # Cost (depth + heuristic)
```

```

def __lt__(self, other):
    return self.cost < other.cost

# Function to display the board in a visually appealing format
def print_board(board):
    print("+-+---+---+")
    for row in range(0, 9, 3):
        row_visual = "|"
        for tile in board[row:row + 3]:
            if tile == 0: # Blank tile
                row_visual += f" {colored(' ', 'cyan')} |"
            else:
                row_visual += f" {colored(str(tile), 'yellow')} |"
        print(row_visual)
    print("+-+---+---+")

# Goal state for the puzzle
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

# Possible moves for the blank tile (up, down, left, right)
moves = {
    'U': -3, # Move up
    'D': 3, # Move down
    'L': -1, # Move left
    'R': 1 # Move right
}

# Function to calculate the heuristic (Manhattan distance)
def heuristic(board):
    distance = 0
    for i in range(9):
        if board[i] != 0:
            x1, y1 = divmod(i, 3)
            x2, y2 = divmod(board[i] - 1, 3)
            distance += abs(x1 - x2) + abs(y1 - y2)
    return distance

# Function to get the new state after a move
def move_tile(board, move, blank_pos):
    new_board = board[:]
    new_blank_pos = blank_pos + moves[move]
    new_board[blank_pos], new_board[new_blank_pos] = new_board[new_blank_pos],
    new_board[blank_pos]
    return new_board

# A* search algorithm
def a_star(start_state):

```

```

open_list = []
closed_list = set()
heapq.heappush(open_list, PuzzleState(start_state, None, None, 0, heuristic(start_state)))

while open_list:
    current_state = heapq.heappop(open_list)

    if current_state.board == goal_state:
        return current_state

    closed_list.add(tuple(current_state.board))

    blank_pos = current_state.board.index(0)

    for move in moves:
        if move == 'U' and blank_pos < 3: # Invalid move up
            continue
        if move == 'D' and blank_pos > 5: # Invalid move down
            continue
        if move == 'L' and blank_pos % 3 == 0: # Invalid move left
            continue
        if move == 'R' and blank_pos % 3 == 2: # Invalid move right
            continue

        new_board = move_tile(current_state.board, move, blank_pos)

        if tuple(new_board) in closed_list:
            continue

        new_state = PuzzleState(new_board, current_state, move, current_state.depth + 1,
                               current_state.depth + 1 + heuristic(new_board))
        heapq.heappush(open_list, new_state)

return None

# Function to print the solution path
def print_solution(solution):
    path = []
    current = solution
    while current:
        path.append(current)
        current = current.parent
    path.reverse()

    for step in path:
        print(f"Move: {step.move}")
        print_board(step.board)

```

```
# Initial state of the puzzle
initial_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]

# Solve the puzzle using A* algorithm
solution = a_star(initial_state)

# Print the solution
if solution:
    print(colored("Solution found:", "green"))
    print_solution(solution)
else:
    print(colored("No solution exists.", "red"))
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

① Hill Climbing:
Algorithm:
 initialize arr[] with the graph points
 curr-value = arr[0]
while \star
 for ($i = 0; i < n; i++$)
 neighbour = arr[$i + 1$]
 if neighbour > curr-value:
 curr-value = neighbour
 return curr-value

Code:

```
import random
```

```
def calculate_attacks(board):
    """Calculates the number of attacking pairs on the board."""
    n = len(board)
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Check row attacks
            if board[i] == board[j]:
                attacks += 1
            # Check diagonal attacks
            elif abs(board[i] - board[j]) == abs(i - j):
                attacks += 1
    return attacks
```

```
def hill_climbing_n_queens(n):
    """Solves the N-Queens problem using Hill Climbing.""""
```

```

# Initialize with a random board
current_board = [random.randint(0, n - 1) for _ in range(n)]
current_attacks = calculate_attacks(current_board)

while current_attacks > 0:
    best_neighbor_board = list(current_board)
    best_neighbor_attacks = current_attacks

    found_better_neighbor = False
    for col_to_move in range(n):
        for row_new_pos in range(n):
            if current_board[col_to_move] == row_new_pos:
                continue # Don't move to the same position

            temp_board = list(current_board)
            temp_board[col_to_move] = row_new_pos
            temp_attacks = calculate_attacks(temp_board)

            if temp_attacks < best_neighbor_attacks:
                best_neighbor_attacks = temp_attacks
                best_neighbor_board = list(temp_board)
                found_better_neighbor = True
            elif temp_attacks == best_neighbor_attacks and not found_better_neighbor:
                # If multiple neighbors have the same best attack count,
                # choose one randomly to avoid always picking the same one
                if random.random() < 0.5: # Simple tie-breaking
                    best_neighbor_board = list(temp_board)

    if not found_better_neighbor:
        # Stuck in a local optimum or reached a solution
        break
    current_board = best_neighbor_board
    current_attacks = best_neighbor_attacks

return current_board, current_attacks

# Example for 4-Queens
n = 4
solution, attacks = hill_climbing_n_queens(n)

print(f"Final board: {solution}")
print(f"Number of attacks: {attacks}")

if attacks == 0:
    print("Solution found!")
else:
    print("Stuck in a local optimum.")

```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

(2)	<u>Simulated Annealing:</u> <u>Algorithm:</u> given: arr[] of temperatures curr-temp = 40 next-temp \leftarrow neighbouring random value $T \leftarrow$ random largest integer while $T > 0$: $\Delta E \leftarrow$ curr-temp - next-temp if $\Delta E > 0$: curr-temp \leftarrow next-temp else: curr-temp \leftarrow next-temp with $p = e^{\Delta E/T}$ decrease T return curr-temp
-----	--

(2)	<u>Simulating annealing</u> 0 0 Q 0 (0)q0q.q0 Q, 0 0 0 0 0 0 Q state.00 0 Q 0 0 (0)00
-----	--

Code:

```
import random
import math

def calculate_attacks(board):
    """Calculates the number of attacking queen pairs on the board."""
    n = len(board)
    attacks = 0
    for i in range(n):
```

```

for j in range(i + 1, n):
    # Check horizontal attacks
    if board[i] == board[j]:
        attacks += 1
    # Check diagonal attacks
    if abs(board[i] - board[j]) == abs(i - j):
        attacks += 1
return attacks

def get_neighbor(board):
    """Generates a neighboring state by randomly moving one queen."""
    n = len(board)
    new_board = list(board)
    queen_to_move = random.randint(0, n - 1)
    new_position = random.randint(0, n - 1)
    new_board[queen_to_move] = new_position
    return tuple(new_board)

def simulated_annealing(n, initial_temperature, cooling_rate, iterations):
    """Solves the N-Queens problem using Simulated Annealing."""
    current_board = tuple(random.randint(0, n - 1) for _ in range(n))
    current_energy = calculate_attacks(current_board)
    best_board = current_board
    best_energy = current_energy
    temperature = initial_temperature

    for i in range(iterations):
        if temperature <= 0:
            break

        neighbor_board = get_neighbor(current_board)
        neighbor_energy = calculate_attacks(neighbor_board)

        # If neighbor is better, accept it
        if neighbor_energy < current_energy:
            current_board = neighbor_board
            current_energy = neighbor_energy
            if current_energy < best_energy:
                best_energy = current_energy
                best_board = current_board
        # If neighbor is worse, accept with a probability
        else:
            probability = math.exp((current_energy - neighbor_energy) / temperature)
            if random.random() < probability:
                current_board = neighbor_board
                current_energy = neighbor_energy

```

```

temperature *= cooling_rate

if current_energy == 0: # Found a solution
    break

return best_board, best_energy

def print_board(board):
    """Prints the N-Queens board."""
    n = len(board)
    for row in range(n):
        line = ["Q" if board[col] == row else "." for col in range(n)]
        print(" ".join(line))

if __name__ == "__main__":
    N = 4 # For the 4-Queens problem
    initial_temp = 100
    cooling_rate = 0.99
    num_iterations = 10000

    solution_board, solution_attacks = simulated_annealing(N, initial_temp, cooling_rate,
num_iterations)

    print(f"Final board configuration: {solution_board}")
    print(f"Number of attacks: {solution_attacks}")

    if solution_attacks == 0:
        print("\nSolution found:")
        print_board(solution_board)
    else:
        print("\nNo perfect solution found within iterations. Best found:")
        print_board(solution_board)

```

Program 6

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

Algorithm:

	KNOWLEDGE BASE						$\alpha \leftarrow \text{query}$	(S)
	FOR PROPOSITIONAL LOGIC						$\neg\neg\alpha \leftarrow \alpha$	
							$\alpha \vee \beta$	
Algorithm:	$\alpha \vee \beta$	$\neg\neg\alpha \leftarrow \alpha$	$\neg\beta \leftarrow \beta$	$\alpha \wedge \beta$	α	β	$\alpha \vee \beta$	$\neg\alpha \rightarrow \beta$
	0 1	1 0	1 1	1 0 1	1 0	1 1	1 1 0	0 1 1
function TT-checkenumeration					0 1	1 1		
return true or false					1 1	0 1		
inputs:	KB \leftarrow Knowledge Base of propositional logic!							
$\alpha \leftarrow$ query symbol	0 1	1 0						
symbols \leftarrow list of propositional logic symbols	0 1	1 0						
return TTcheckall (KB, α , symbols)	1 0 0							
	0 0 0	1 1	1 0	0 0 0				
function TTcheckall return true or false								
if Empty?								
symbols (true)	0 0 1	1 1 0	0 1 1	1 0 0				
if true then return True? (α ; model)	0 1 0	1 0 1	0 0 0	1 1 1				
else return \neg True	1 0 1	0 1 0	1 1 1	0 0 0				
else do								
P \leftarrow First(symbols)	0 0 0	1 1 1	0 1 0	1 0 1				
rest \leftarrow Rest(symbols)	0 0 1	1 1 0	0 1 1	1 0 0				
return TTcheckall (KB, α) $\cup \{P = \text{True}\}$	0 1 0	1 0 1	0 0 0	1 1 1				
and TTcheckall (KB, α) $\cup \{P = \text{False}\}$	1 0 0	0 1 1	1 1 0	0 0 0				
model, symbols	0 0 0	1 1 1	0 1 0	1 0 1				
Output: True \rightarrow if KB $\models \alpha$	1 0 0							
false \rightarrow otherwise	1 1 0							

T-SAT								
(2)	$R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	$P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ $Q \vee R$							
	T-SAT							
	P Q R $R \rightarrow P$ $P \rightarrow \neg Q$ <							

Code:

```
import pandas as pd
import itertools

# Define propositions
props = ['P', 'Q', 'R']

# Generate all truth assignments for P, Q, R
truth_values = list(itertools.product([False, True], repeat=3))

def implies(a, b):
    return (not a) or b

# Evaluate KB sentences and entailments
rows = []
for (P, Q, R) in truth_values:
    sentence1 = implies(Q, P)      # Q -> P
    sentence2 = implies(P, not Q)   # P -> ¬Q
    sentence3 = Q or R             # Q ∨ R

    kb_true = sentence1 and sentence2 and sentence3

    entail_R = R
    entail_R_implies_P = implies(R, P)
    entail_Q_implies_R = implies(Q, R)

    rows.append({
        'P': P, 'Q': Q, 'R': R,
        'Q->P': sentence1,
        'P->¬Q': sentence2,
        'Q ∨ R': sentence3,
        'KB True': kb_true,
        'Entail R': entail_R,
        'Entail R->P': entail_R_implies_P,
        'Entail Q->R': entail_Q_implies_R
    })

# Create DataFrame
df = pd.DataFrame(rows)

# Filter models where KB is true
models_where_KB_true = df[df['KB True']]

# Print full truth table
print("Full Truth Table:\n")
```

```

print(df.to_string(index=False))

# Checking entailments (whether KB entails each sentence)
# KB entails a sentence if the sentence is true in all models where KB is true
entail_R_result = models_where_KB_true['Entail R'].all()
entail_R_implies_P_result = models_where_KB_true['Entail R->P'].all()
entail_Q_implies_R_result = models_where_KB_true['Entail Q->R'].all()

print("\nModels where KB is True:")
print(models_where_KB_true.to_string(index=False))

print("\nEntailment Results:")
print(f"Does KB entail R? {'Yes' if entail_R_result else 'No'}")
print(f"Does KB entail R -> P? {'Yes' if entail_R_implies_P_result else 'No'}")
print(f"Does KB entail Q -> R? {'Yes' if entail_Q_implies_R_result else 'No'}")

```

Program 7

Implement unification in first order logic

Algorithm:

zoholas		LAB 8	SV A = 70	(S)
24	35 40 35 A	3 8 A		
3	Unification Algorithm:	0 0 0		
4	0 0 0	1 0 0		
5	1: If ψ_1 or ψ_2 is a variable or constant then:			
6	a. If ψ_1 & ψ_2 are identical then return NIL			
7	b. Else if ψ_1 is a variable then:			
8	i. If ψ_1 occurs in ψ_2 then: return FAILURE			
9	ii. Else return $\{\psi_1 / \psi_2\}$			
10	iii. Else if ψ_2 is a variable then:			
11	If ψ_2 occurs in ψ_1 then: return FAILURE			
12	Else return $\{\psi_1 / \psi_2\}$			
13	d. Else return FAILURE			
14	2: If initial predicate symbols ψ_1 & ψ_2 are not same then return FAILURE			
15	3: If ψ_1 & ψ_2 have different no. of arguments then, return FAILURE			
16	4: Set substitution set(SUBST) to NIL			
17	5: For i=1 to no. of elements in ψ_1 :			
18	→ call unification function with i^{th} ele of ψ_1 & i^{th} ele of ψ_2 and put the result			
19	→ If S=FAIL return FAILURE			
20	→ If S ≠ NIL then do:			
21	a) Apply S to the remainder of both L1 & L2			
22	b) SUBST = APPEND(S, SUBST)			
23	6: Return SUBST			

①	P(f(x), g(y), z)	Date: 2023-07-17
	P(f(g(z)), g(f(a)), f(a))	Initials: f, g, P, a, z
	P = P	(a + no. of args) \rightarrow v
	same number of arguments	initials with numbers
	I f(x) f(g(z)) \Rightarrow x g(z)	
	g(f(a)) g(y) \Rightarrow f(a) y \rightarrow f(a) \rightarrow v	
	y f(a) \rightarrow y f(a) \rightarrow v	
	∴ Unifiable $\Theta = \{x g(z), y f(a)\}$	
②	S(x, f(x))	(x a) \rightarrow v
	S(f(y), y)	v \rightarrow v
	I f(x) f(y) \rightarrow (a a) \rightarrow v	
	f(f(y)) f(f(a)) \Rightarrow f(y) f(a) \rightarrow v	
	f(f(y)) f(f(a)) \Rightarrow f(y) f(a) \rightarrow v	
	f(f(y)) f(f(a)) \Rightarrow f(y) f(a) \rightarrow v	
	f(f(y)) f(f(a)) \Rightarrow f(y) f(a) \rightarrow v	
③	R(x, g(x))	(v a) \rightarrow v
	R(g(y), g(g(z)))	v \rightarrow v
	x g(y) \rightarrow (v b, v c, v d) \rightarrow v	
	g(y) g(g(z)) \Rightarrow g x g(z) \rightarrow v	
	g(y) g(g(z)) \Rightarrow g x g(z) \rightarrow v	
	g(y) g(g(z)) \Rightarrow g x g(z) \rightarrow v	
	g(y) g(g(z)) \Rightarrow g x g(z) \rightarrow v	
	∴ Unifiable $\Theta = \{y z\}$	

Code:

```
import re

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

def parse_term(term):
    # Parses a function or constant or variable into structured form
    term = term.strip()
    if '(' not in term:
        return term
    functor, args_str = term.split('(', 1)
    args = []
    depth = 0
```

```

current = ""
for ch in args_str[:-1]: # skip last ')'
    if ch == ',' and depth == 0:
        args.append(parse_term(current.strip()))
        current = ""
    else:
        if ch == '(':
            depth += 1
        elif ch == ')':
            depth -= 1
        current += ch
if current:
    args.append(parse_term(current.strip()))
return (functor.strip(), args)

def occurs_check(var, x, subst):
    if var == x:
        return True
    elif isinstance(x, str):
        if x in subst:
            return occurs_check(var, subst[x], subst)
        return False
    elif isinstance(x, tuple):
        return any(occurs_check(var, arg, subst) for arg in x[1])
    return False

def substitute(x, subst):
    if isinstance(x, str):
        # If x is a variable and has a substitution, apply it recursively.
        # This handles chained substitutions like x -> y, y -> f(z).
        if x in subst:
            return substitute(subst[x], subst)
        else:
            # If it's not in the substitution, it's either a constant string
            # or a variable that hasn't been substituted yet.
            return x
    elif isinstance(x, tuple):
        functor, args = x
        return (functor, [substitute(arg, subst) for arg in args])
    return x

def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    x = substitute(x, subst)
    y = substitute(y, subst)
    if x == y:

```

```

        return subst
    elif is_variable(x):
        if occurs_check(x, y, subst):
            raise ValueError(f"Occurs check failed for {x} in {y}")
        subst[x] = y
        return subst
    elif is_variable(y):
        if occurs_check(y, x, subst):
            raise ValueError(f"Occurs check failed for {y} in {x}")
        subst[y] = x
        return subst
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            raise ValueError(f"Functor mismatch: {x[0]} vs {y[0]}")
        for a, b in zip(x[1], y[1]):
            subst = unify(a, b, subst)
        return subst
    else:
        raise ValueError(f"Cannot unify {x} with {y}")

# ----- Test -----
x = parse_term("P(f(x), g(y), y)")
y = parse_term("P(f(g(z)), g(f(a)), f(a))")

print("Parsed terms:")
print("x =", x)
print("y =", y)

try:
    result = unify(x, y)
    print("\nUnification successful!")
    print("Substitution set:")
    for var, val in result.items():
        print(f" {var} = {val}")
except ValueError as e:
    print("\nUnification failed:", e)

```

Program 8

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

Algorithm:

10.	CNF form of Q + (assumption) rules
	$\forall x \neg \forall y \neg (\text{Animal}(y) \vee \text{Loves}(x, y)) \vee \exists y \text{Loves}(y, x)$
	$\forall x \neg \forall y (\neg \text{Animal}(y) \wedge \neg \text{Loves}(x, y)) \vee \exists y \text{Loves}(y, x)$
	$\forall x \exists y (\text{Animal}(y) \vee \text{Loves}(x, y)) \vee \exists y \text{Loves}(y, x)$
	$\forall x \exists y \exists z [\text{Animal}(y) \vee \text{Loves}(x, y) \vee \text{Loves}(z, x)]$
	$\forall x [\text{Animal}(g(x)) \vee \text{Loves}(x, g(x)) \vee \text{Loves}(f(x), x)]$
	$\text{Animal}(g(x)) \vee \text{Loves}(x, g(x)) \vee \text{Loves}(f(x), x)$
11.	Knowledge Base
	$\text{Man}(\text{Marcus})$
	$\text{Pompeian}(\text{Marcus})$
	$\forall x (\text{Pompeian}(x) \rightarrow \text{Roman}(x))$
	$\forall x (\text{Roman}(x) \rightarrow \text{Loyal}(x))$
	$\forall x (\text{Man}(x) \rightarrow \text{Person}(x))$
	$\forall x (\text{Person}(x) \rightarrow \text{Mortal}(x))$
	Query $\text{mortal}(\text{marcus})$
	Sol: $\forall x (\text{Pompeian}(x) \rightarrow \text{Roman}(x)) \text{ and } \text{Pompeian}(\text{marcus})$
	$\Rightarrow \text{Roman}(\text{marcus})$
	$\forall x (\text{Roman}(x) \rightarrow \text{Loyal}(x)) \text{ and } \text{Roman}(\text{marcus})$
	$\Rightarrow \text{Loyal}(\text{marcus})$
	$\forall x (\text{man}(x) \rightarrow \text{Person}(x)) \text{ and } \text{man}(\text{marcus})$
	$\Rightarrow \text{Person}(\text{marcus})$
	$\forall x (\text{Person}(x) \rightarrow \text{Mortal}(x)) \text{ and } \text{person}(\text{marcus})$
	$\Rightarrow \text{mortal}(\text{marcus})$

man (marcus)	$\forall x [\text{man}(x) \rightarrow \text{Person}(x)]$
	$\text{Person}(\text{marcus})$
	$\forall x [\text{Person}(x) \rightarrow \text{Mortal}(x)]$
	$\text{mortal}(\text{marcus})$
	$(\text{man}(\text{marcus}) \wedge \forall x [\text{man}(x) \rightarrow \text{Person}(x)]) \rightarrow \text{Person}(\text{marcus})$
	$(\text{Person}(\text{marcus}) \wedge \forall x [\text{Person}(x) \rightarrow \text{Mortal}(x)]) \rightarrow \text{mortal}(\text{marcus})$
	$(\text{man}(\text{marcus}) \wedge \forall x [\text{man}(x) \rightarrow \text{Person}(x)] \wedge \forall x [\text{Person}(x) \rightarrow \text{Mortal}(x)]) \rightarrow \text{mortal}(\text{marcus})$
	$(\text{man}(\text{marcus}) \wedge \forall x [\text{man}(x) \rightarrow \text{Person}(x)] \wedge \forall x [\text{Person}(x) \rightarrow \text{Mortal}(x)]) \rightarrow \text{mortal}(\text{marcus})$

Code:

```
import re

# -----
# Helper functions
# -----

def eliminate_implications(expr):
    """Eliminate implications (A -> B becomes ~A | B)."""
    expr = re.sub(r'\((.*?))->\((.*?))', r'(\1)|\2)', expr)
    return expr

def move_not_inwards(expr):
    """Apply De Morgan's laws and double negation elimination."""
    expr = expr.replace("~~", "")
    expr = expr.replace("~(A&B)", "(~A|~B)")
    expr = expr.replace("~(A|B)", "(~A&~B)")
    return expr

def standardize_variables(expr):
    """Rename variables to avoid clashes."""
    # For demo, just ensure lowercase vars get unique suffixes
    var_map = {}
    count = 0
    new_expr = ""
    for ch in expr:
        if ch.islower() and ch.isalpha():
            if ch not in var_map:
                var_map[ch] = chr(ord('a') + count)
                count += 1
            new_expr += var_map[ch]
        else:
            new_expr += ch
    return new_expr

def skolemize(expr):
    """Replace existential quantifiers with Skolem constants/functions."""
    expr = re.sub(r'\exists [a-z]\.', "", expr) # Remove existential quantifier
    return expr.replace("x", "c") # Replace var with Skolem constant (simple demo)

def drop_universal(expr):
    """Remove universal quantifiers (implicit in CNF)."""
    expr = re.sub(r'\forall [a-z]\.', "", expr)
    return expr

def distribute_or_over_and(expr):
```

```

"""Simplified distribution (only handles basic patterns)."""
# In a full implementation you'd need a tree structure, not regex.
return expr.replace("|", "\vee").replace("&", "\wedge")

# -----
# Main CNF conversion function
# -----


def fol_to_cnf(expr):
    print("Original:", expr)
    expr = eliminate_implications(expr)
    print("→ No implications:", expr)
    expr = move_not_inwards(expr)
    print("→ Negations inward:", expr)
    expr = standardize_variables(expr)
    print("→ Standardized vars:", expr)
    expr = skolemize(expr)
    print("→ Skolemized:", expr)
    expr = drop_universal(expr)
    print("→ Dropped universals:", expr)
    expr = distribute_or_over_and(expr)
    print("→ CNF form:", expr)
    return expr

# -----
# Example usage
# -----


formula = "∀ x.(P(x) -> ∃ y.(Q(y) & R(x,y)))"
cnf = fol_to_cnf(formula)
print("\nFinal CNF:", cnf)

# Simple Forward Reasoning (Forward Chaining) in FOL

# Knowledge Base (Facts + Rules)
# Rules are written in the form: (premises, conclusion)
# Facts are just single statements

def forward_chaining(kb, query):
    facts = set()
    rules = []

    # Separate facts and rules
    for statement in kb:
        if "=>" in statement:
            premises, conclusion = statement.split("=>")
            premises = set(p.strip() for p in premises.split("&"))

    facts.add(query)

```

```

        rules.append((premises, conclusion.strip()))
else:
    facts.add(statement.strip())

print("Initial Facts:", facts)
print("Rules:", rules)
print("Query:", query)

inferred = True
while inferred:
    inferred = False
    for premises, conclusion in rules:
        # Check if all premises are in facts
        if premises.issubset(facts) and conclusion not in facts:
            facts.add(conclusion)
            print(f"Inferred: {conclusion}")
            inferred = True
            if conclusion == query:
                print("Query proven!")
                return True

print("Query cannot be proven.")
return False

# Example Knowledge Base
kb = [
    "A",                      # Fact
    "A & B => C",          # Rule 1
    "C => D",                # Rule 2
    "D & E => F",          # Rule 3
    "B"                      # Fact
]
query = "D"

# Run forward chaining
forward_chaining(kb, query)

```

Program 9

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

Algorithm:

$(x \text{ likes}(x)) \vee (x \text{ boot}) \vdash \text{RESOLUTION}$	
For start?	
Algorithm:	
$\neg S \vee (S) \text{ boot} \vdash (\neg S) \text{ boot} \vdash \text{ (Answer)}$	
1: Convert all sentences to CNF	
2: Negate conclusion $S \neg \&$ Convert result to CNF	
3: Add negated conclusion ($\neg S$) to the premise clauses	
4: Repeat until contradiction or no progress is made	
a. Select 2 clauses (call them parent clauses)	
b. Resolve them together, performing all required unifications	
c. If resolvent is the empty clause, a contradiction has been found	
d. If not, add resolvent to premises	
Example: having result	
Premises: $\neg \text{food}(x) \vee \text{likes}(\text{John}, x)$	
$\text{food}(\text{Apple})$	
$\text{food}(\text{vegetables})$	
$\neg \text{eats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$	
$\text{eats}(\text{Anil}, \text{Peanuts})$	
$\text{alive}(\text{Anil})$	
$\neg \text{eats}(\text{Anil}, w) \vee \text{eats}(\text{Harry}, w)$	
$\text{killed}(g) \vee \text{alive}(g)$	
$\neg \text{alive}(k) \vee \neg \text{killed}(k)$	
$\text{likes}(\text{John}, \text{Peanuts})$	

M	T	W	T	F	S	S
					Page No.:	YOUVA
				Date:		

Resolution:

$\neg \text{likes}(\text{John}, \text{Peanuts}) \quad \neg \text{food}(x) \vee \text{likes}(\text{John}, x)$

$\neg \text{Food}(\text{Peanuts}) \quad \neg \text{eats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$

$\neg \text{eats}(y, \text{Peanuts}) \vee \text{killed}(y) \quad \neg \text{eats}(\text{Anil}, \text{peanuts})$

$\neg \text{killed}(\text{Anil}) \quad \neg \text{alive}(k) \vee \neg \text{killed}(k)$

$\neg \text{alive}(\text{Anil}) \quad \text{alive}(\text{Anil})$

{ } Hence proved

Code:

```
# Simple Forward Reasoning (Forward Chaining) in FOL
# Knowledge Base (Facts + Rules)
# Rules are written in the form: (premises, conclusion)
# Facts are just single statements
```

```
def forward_chaining(kb, query):
    facts = set()
    rules = []

    # Separate facts and rules
    for statement in kb:
        if "=>" in statement:
            premises, conclusion = statement.split("=>")
            premises = set(p.strip() for p in premises.split("&"))
            rules.append((premises, conclusion.strip()))
        else:
            facts.add(statement.strip())
```

```

print("Initial Facts:", facts)
print("Rules:", rules)
print("Query:", query)

inferred = True
while inferred:
    inferred = False
    for premises, conclusion in rules:
        # Check if all premises are in facts
        if premises.issubset(facts) and conclusion not in facts:
            facts.add(conclusion)
            print(f"Inferred: {conclusion}")
            inferred = True
        if conclusion == query:
            print("Query proven!")
            return True

print("Query cannot be proven.")
return False

# Example Knowledge Base
kb = [
    "A",           # Fact
    "A & B => C",   # Rule 1
    "C => D",       # Rule 2
    "D & E => F",   # Rule 3
    "B"           # Fact
]
query = "D"

# Run forward chaining
forward_chaining(kb, query)

Resolution:
!pip install sympy

from sympy import symbols, Or, Not, And
from sympy.logic.inference import satisfiable

# Define propositional atoms (grounded)
food_Apple, food_Veg, food_Peanuts = symbols('food_Apple food_Veg food_Peanuts')
likes_John_Apple, likes_John_Veg, likes_John_Peanuts = symbols('likes_John_Apple'
                                                               'likes_John_Veg likes_John_Peanuts')
eats_Anil_Peanuts, killed_Anil, alive_Anil = symbols('eats_Anil_Peanuts killed_Anil alive_Anil')


```

```

# --- Knowledge Base (grounded CNF clauses) ---
KB = [
    Or(Not(food_Peanuts), likes_John_Peanuts),    # a.  $\neg$ food(Peanuts)  $\vee$  likes(John, Peanuts)
    food_Apple,                                # b. food(Apple)
    food_Veg,                                   # c. food(vegetables)
    Or(Not(eats_Anil_Peanuts), killed_Anil, food_Peanuts), # d.  $\neg$ eats(Anil, Peanuts)  $\vee$  killed(Anil)
    \ food(Peanuts)
    eats_Anil_Peanuts,                      # e. eats(Anil, Peanuts)
    alive_Anil,                            # f. alive(Anil)
    Or(killed_Anil, Not(alive_Anil)),      # h. killed(Anil)  $\vee$   $\neg$ alive(Anil)
    Or(Not(alive_Anil), Not(killed_Anil))  # i.  $\neg$ alive(Anil)  $\vee$   $\neg$ killed(Anil)
]
# --- Negated conclusion ---
negated_conclusion = Not(likes_John_Peanuts)
KB.append(negated_conclusion)

# --- Check satisfiability ---
result = satisfiable(And(*KB))

if result is False:
    print("Conclusion PROVED by resolution: likes(John, Peanuts)")
else:
    print("Conclusion NOT proved.")
    print("Satisfying model (means no contradiction):")
    print(result)

```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

ALPHA-BETA PRUNING:

```
Function Alpha-Beta-Search (state) returns an action
    v ← MAX-VALUE (state, -∞, +∞)
    return the action in ACTIONS (state) with value v

Function MAX-VALUE (state; α, β) returns a utility value
    if TERMINAL-TEST (state) then return UTILITY (state)
    v ← -∞
    for each a in ACTIONS (state) do
        v ← MAX (v, MIN-VALUE (RESULT (s, a), α, β))
        if v ≥ β then return v
        α ← MAX (α, v)
    return v

Function MIN-VALUE (state, α, β) returns a utility value
    if TERMINAL-TEST (state) then return UTILITY (state)
    v ← +∞
    for each a in ACTIONS (state) do
        v ← MIN (v, MAX-VALUE (RESULT (s, a), α, β))
        if v ≤ α then return v
        β ← MIN (β, v)
    return v

Output: Initial State : [1, 2, 0, 3, 4, 5, 6, 7, 8]
Initial Depth: 5
Best Value found: 11
The return of the alpha-beta pruning is: 11
```

Code:

```
import math

def alpha_beta_pruning(state, depth, alpha, beta, maximizing_player):
    # Base case: leaf node or maximum depth reached
    if depth == 0 or is_goal_state(state):
        return heuristic_value(state)
```

```

if maximizing_player:
    max_eval = -math.inf
    for child_state in get_possible_moves(state):
        eval = alpha_beta_pruning(child_state, depth - 1, alpha, beta, False)
        max_eval = max(max_eval, eval)
        alpha = max(alpha, eval)
        if beta <= alpha:
            break # Beta cutoff
    return max_eval
else: # Minimizing player
    min_eval = math.inf
    for child_state in get_possible_moves(state):
        eval = alpha_beta_pruning(child_state, depth - 1, alpha, beta, True)
        min_eval = min(min_eval, eval)
        beta = min(beta, eval)
        if beta <= alpha:
            break # Alpha cutoff
    return min_eval

def is_goal_state(state):
    """Checks if the current state is the solved 8-puzzle."""
    # Assuming the goal state is a flattened list [1, 2, 3, 4, 5, 6, 7, 8, 0]
    return state == [1, 2, 3, 4, 5, 6, 7, 8, 0]

def heuristic_value(state):
    """Calculates the Manhattan distance heuristic for the 8-puzzle."""
    goal_state = {
        1: (0, 0), 2: (0, 1), 3: (0, 2),
        4: (1, 0), 5: (1, 1), 6: (1, 2),
        7: (2, 0), 8: (2, 1), 0: (2, 2)
    }
    manhattan_distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i * 3 + j]
            if tile != 0:
                goal_pos = goal_state[tile]
                manhattan_distance += abs(i - goal_pos[0]) + abs(j - goal_pos[1])
    return manhattan_distance

def get_possible_moves(state):
    """Returns a list of all possible next states from the current state."""
    possible_moves = []
    zero_index = state.index(0)
    zero_row, zero_col = divmod(zero_index, 3)

    # Define possible movements (up, down, left, right)

```

```

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

for move_row, move_col in movements:
    new_row, new_col = zero_row + move_row, zero_col + move_col

    # Check if the new position is within the board boundaries
    if 0 <= new_row < 3 and 0 <= new_col < 3:
        new_state = list(state) # Create a mutable copy of the state
        swap_index = new_row * 3 + new_col
        # Swap the blank tile with the tile at the new position
        new_state[zero_index], new_state[swap_index] = new_state[swap_index],
        new_state[zero_index]
        possible_moves.append(new_state)

return possible_moves
initial_state = [1, 2, 0, 3, 4, 5, 6, 7, 8]
initial_depth = 5

print(f"Initial State: {initial_state}")
print(f"Initial Depth: {initial_depth}")
best_value = alpha_beta_pruning(initial_state, initial_depth, -math.inf, math.inf, True)
print(f"Best value found: {best_value}")
print(f"The result of the alpha-beta pruning (best value from the initial state) is: {best_value}")

```

Minimax:

```

import math

# Represents the players
X = 'X'
O = 'O'
EMPTY = ''

def check_winner(board):
    """
    Checks if there's a winner on the board.
    Returns X if X wins, O if O wins, None if no winner yet, or 'Tie' if full and no winner.
    """
    # Check rows
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != EMPTY:
            return row[0]
    # Check columns
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != EMPTY:
            return board[0][col]
    # Check diagonals

```

```

if board[0][0] == board[1][1] == board[2][2] and board[0][0] != EMPTY:
    return board[0][0]
if board[0][2] == board[1][1] == board[2][0] and board[0][2] != EMPTY:
    return board[0][2]

# Check for tie (no winner and board is full)
if all(cell != EMPTY for row in board for cell in row):
    return 'Tie'

return None

def minimax(board, is_maximizing_player):
    """
    Implements the Minimax algorithm to find the optimal move.
    is_maximizing_player is True for 'X' (AI), False for 'O' (opponent).
    """
    winner = check_winner(board)
    if winner == X:
        return 1 # X wins, maximizing player gets a high score
    elif winner == O:
        return -1 # O wins, minimizing player gets a low score
    elif winner == 'Tie':
        return 0 # Tie game

    if is_maximizing_player:
        best_score = -math.inf
        for r in range(3):
            for c in range(3):
                if board[r][c] == EMPTY:
                    board[r][c] = X # Make the move
                    score = minimax(board, False)
                    board[r][c] = EMPTY # Undo the move
                    best_score = max(best_score, score)
        return best_score
    else: # Minimizing player
        best_score = math.inf
        for r in range(3):
            for c in range(3):
                if board[r][c] == EMPTY:
                    board[r][c] = O # Make the move
                    score = minimax(board, True)
                    board[r][c] = EMPTY # Undo the move
                    best_score = min(best_score, score)
        return best_score

def find_best_move(board):
    """
    """

```

```

Finds the best move for the AI (X) using the Minimax algorithm.
"""

best_score = -math.inf
best_move = None

for r in range(3):
    for c in range(3):
        if board[r][c] == EMPTY:
            board[r][c] = X # Try the move
            score = minimax(board, False) # Evaluate the board state
            board[r][c] = EMPTY # Undo the move

            if score > best_score:
                best_score = score
                best_move = (r, c)
return best_move

def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    for row in board:
        print("|".join(row))
    print("----")

# Example usage:
if __name__ == "__main__":
    # Initial empty board
    board = [[EMPTY, EMPTY, EMPTY],
              [EMPTY, EMPTY, EMPTY],
              [EMPTY, EMPTY, EMPTY]]

    current_player = X

    while check_winner(board) is None:
        print_board(board)
        if current_player == X:
            print("AI's turn (X)")
            move = find_best_move(board)
            if move:
                board[move[0]][move[1]] = X
            else:
                print("No valid moves left for AI.")
                break
        else:
            print("Human's turn (O)")
            while True:
                try:
                    row = int(input("Enter row (0-2): "))

```

```
col = int(input("Enter column (0-2): "))
if 0 <= row <= 2 and 0 <= col <= 2 and board[row][col] == EMPTY:
    board[row][col] = O
    break
else:
    print("Invalid move. Try again.")
except ValueError:
    print("Invalid input. Please enter numbers.")

current_player = O if current_player == X else X

print_board(board)
final_result = check_winner(board)
if final_result == 'Tie':
    print("It's a Tie!")
elif final_result:
    print(f"Player {final_result} wins!")
else:
    print("Game ended unexpectedly.")
```