

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



LAB REPORT
on
Artificial Intelligence (23CS5PCAIN)

Submitted by

Makadia Rishit Dilipbhai (1BM23CS177)

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 **Makadia Rishit Dilipbhai (1BM23CS177)**, 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.

Dr. K.R. Mamatha Associate/Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
---	--

Index

Sl. No.	Date	Experiment Title	Page No.
1	22-8-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4
2	29-8-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	10
3	12-9-2025	Implement A* search algorithm	13
4	19-9-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	17
5	3-10-2025	Simulated Annealing to Solve 8-Queens problem	20
6	17-10-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	23
7	24-10-2025	Implement unification in first order logic	26
8	31-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	28
9	7-11-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	30
10	14-11-2025	Implement Alpha-Beta Pruning.	33

Github Link:

https://github.com/rishitmakadia/1BM23CS177_AIN_5

Program 1

Implement Tic - Tac - Toe Game
Implement vacuum cleaner agent

Algorithm:

<p>Lab 2) Implement Tic-Tac-Toe Game * Step-By-Step</p> <p>Algorithm:</p> <pre> Step 1: Start Step 2: Take user symbol input 'X' initialize matrix T[3][3] = { } Step 3: Take input coordinates for user symbol (x,y) check (int i & int j == 2) if (matrix[i][j] == ' ') checkWin (symbol) Go To Step 4 else "Input field is already filled cout << "Draw" Step 4: checkWin (int symbol) for i=0 to 2 for j=0 to 2 if (matrix[i][j] == symbol && matrix[i][j+1] == symbol && matrix[i][j+2] == symbol) // check row return true if (matrix[i][j] == symbol && matrix[i+1][j] == symbol && matrix[i+2][j] == symbol) // check column return true if (matrix[i][j] == symbol && matrix[i+1][j+1] == symbol && matrix[i+2][j+2] == symbol) // check diagonal return true if (matrix[i][j] == symbol && matrix[i+1][j-1] == symbol && matrix[i+2][j-2] == symbol) // check anti-diagonal return true return false Step 5: if checkWin == true User wins else bot turn() Go To Step 6 </pre>	<p>Step 6: $x = \text{rand}(0, 3), y = \text{rand}(0, 3)$ // for AI logic Step 7 check (matrix[i][j] == ' ') symbol Call to Step 3.</p> <p>Step 7: Step 6! Not Cond (symbol) // main using Minimax algorithm (back-tracking) priority list of moves: (0,0) black square → Not Minimise opponent's best move → Strategic position → Randomly It moves through all empty spots, makes a move, it returns the ans based on priority.</p>	<p>Algorithm:</p> <p>Bot Logic</p> <ul style="list-style-type: none"> There are 8 ways to win (3 rows + 3 cols + 2 diagonals) My bot logic checks for these 8 ways based on priority If the loop ends <ul style="list-style-type: none"> If we have no winning conditions satisfying resulting in draw. Or either bot or human wins <p>Output:</p>
--	--	--

<p>Lab 4) Implement Vacuum Cleaner Agent</p> <p>Algorithm:</p> <pre> Step 1: Start Step 2: Define matrix for dirt (coordinates) for cleaner to move. Step 3: if it encounters dirt it cleans it & moves straight else if it encounters a wall it checks left or right & move in that direction Step 4: When the room is clean (i.e. matrix == 0) Step 5: Stop. </pre>	<p>Output:</p> <p>$\begin{bmatrix} \downarrow & \downarrow \\ 0 & 0 \end{bmatrix}$ // NC starts from a starting point in a matrix 'd'</p> <p>$\begin{bmatrix} \downarrow & \downarrow \\ 0 & X \end{bmatrix}$ // wall encountered so turn</p> <p>$\begin{bmatrix} \downarrow & \downarrow \\ 0 & * \end{bmatrix}$ // Dirt encountered, so clean</p> <p>$\begin{bmatrix} \downarrow & \downarrow \\ 0 & 0 \end{bmatrix}$ // wall encountered so turn // final reached</p> <p>$\begin{bmatrix} \downarrow & \downarrow \\ 0 & 0 \end{bmatrix}$ // back on initial point</p>
---	--

Code:

```
Tic - Tac - Toe
print('tic_tac_toe')
board={'1':' ', '2':' ', '3':' ',
       '4':' ', '5':' ', '6':' ',
       '7':' ', '8':' ', '9':' '}
}

def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('---')
    print(board[4] + ' |' + board[5] + ' |' + board[6])
    print('---')
    print(board[7] + ' |' + board[8] + ' |' + board[9])
    print('\n')

def spaceFree(pos):
    if(board[pos]==' '):
        return True
    else:
        return False

def checkWin():
    if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
        return True
    elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
        return True
    elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
        return True
    elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
        return True
    elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
        return True
    elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
        return True
    elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
        return True
    elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
        return True
    else:
        return False

def checkMoveForWin(move):
    if (board[1]==board[2] and board[1]==board[3] and board[1]==move):
        return True
```

```

        elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
            return True
        elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
            return True
        elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
            return True
        elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
            return True
        elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
            return True
        elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
            return True
        elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
            return True
        else:
            return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True

def insertLetter(letter, position):
    if (spaceFree(position)):
        board[position] = letter
        printBoard(board)
        if (checkDraw()):
            print('Draw!')
        elif (checkWin()):
            if (letter == 'X'):
                print('Bot wins!')
            else:
                print('You win!')
        return
    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)
        return

player = 'O'
bot ='X'

def playerMove():
    position=int(input('Enter position for O:'))
    insertLetter(player, position)

```

```

return

def compMove():
    bestScore=-1000
    bestMove=0
    for key in board.keys():
        if (board[key]==' '):
            board[key]=bot
            score = minimax(board, False)
            board[key] = ''
            if (score > bestScore):
                bestScore = score
                bestMove = key
    insertLetter(bot, bestMove)
    return

def minimax(board, isMaximizing):
    if (checkMoveForWin(bot)):
        return 1
    elif (checkMoveForWin(player)):
        return -1
    elif (checkDraw()):
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == '':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ''
                if (score > bestScore):
                    bestScore = score
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == '':
                board[key] = player
                score = minimax(board, True)
                board[key] = ''
                if (score < bestScore):
                    bestScore = score
        return bestScore

while not checkWin():
    compMove()

```

```
playerMove()
```

```
Vacuum-Cleaner
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()
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:

The image shows two pages of handwritten notes. The left page is titled 'Lab 3' and contains the algorithm steps. The right page shows the execution of the algorithm with a starting state and three iterations (Iteration 1, Iteration 2, Iteration 3) leading to a final state.

Lab 3 Implement Iterative deepening search algorithm

Algorithm:

- Step 1: START**
- Step 2: Initialize**
depth_limit = 0, iteration = 1.
- Step 3: Iterative Deepening Loop**
 - Initialize visited matrix.
 - last_visited_state = start_state.
 - Do DLS until current state = goal state.
 - depth limit = path length + path initialized with visited.
- Step 4: DLS Recursive steps**
 - Update last_visited_state with current state.
 - If current state == goal return path step 5.
 - If successor not in visited
 - successor as current state
 - decrease depth.
 - update path with successor appended.
 - current visited.
 - remove current from visited (backtracking)
- Step 5: Finish the current depth.**
 - Print iteration.
 - If DLS found path step 7.

Output:

Starting state : 1 2 3
4 0 6
7 5 8

Iteration 1 Depth limit=0
1 2 3
4 6 0
7 5 8

Iteration 2 Depth limit=1
1 2 3
4 6 0
7 5 8

Iteration 3 Depth limit=2
1 2 3
4 5 6
7 8 0

Final state : 1 2 3
4 5 6
7 8 0

Code:

```
// IDFS

from collections import deque

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

MOVES = {
    'up': -3,
    'down': 3,
    'left': -1,
    'right': 1
}

def is_valid_move(blank_idx, move):
    if move == 'left' and blank_idx % 3 == 0:
        return False
```

```

if move == 'right' and blank_idx % 3 == 2:
    return False
if move == 'up' and blank_idx < 3:
    return False
if move == 'down' and blank_idx > 5:
    return False
return True

def get_successors(state):
    successors = []
    blank_idx = state.index(0)

    for move in MOVES:
        if is_valid_move(blank_idx, move):
            swap_idx = blank_idx + MOVES[move]
            new_state = list(state)
            new_state[blank_idx], new_state[swap_idx] = new_state[swap_idx], new_state[blank_idx]
            successors.append(tuple(new_state))
    return successors

def DLS(state, goal, limit, path, visited, last_state_holder):
    last_state_holder[0] = state

    if state == goal:
        return path
    if limit == 0:
        return None

    visited.add(state)

    for successor in get_successors(state):
        if successor not in visited:
            result = DLS(successor, goal, limit - 1, path + [successor], visited, last_state_holder)
            if result is not None:
                return result

    visited.remove(state)
    return None

def IDDFS(start, goal):
    depth = 0
    iteration = 1
    while True:
        visited = set()
        last_state_holder = [start]
        path = DLS(start, goal, depth, [start], visited, last_state_holder)

```

```

print(f"Iteration {iteration} completed at Depth limit = {depth}")
print("Last visited puzzle state in this iteration:")
print_puzzle(last_state_holder[0])

if path is not None:
    return path
depth += 1
iteration += 1

def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

if __name__ == "__main__":
    start_state = (1, 2, 4,
                   0, 8, 6,
                   7, 5, 3)

    print("Starting state:")
    print_puzzle(start_state)

    solution_path = IDDFS(start_state, GOAL_STATE)

    print(f"Solution found in {len(solution_path) - 1} moves:")
    for step in solution_path:
        print_puzzle(step)

```

Program 3

Implement A* search algorithm

Algorithm:

Implement Hill Climbing Search.

Lab-5a Implement A* algorithm using 8 puzzle.

Algorithm:

Step 1: Start

Step 2: Create a Goal Matrix (User choice)
Ex- Assign the first 8 blocks A,B,C,...H respectively. Map each number to alphabets.
 $A \rightarrow 1, B \rightarrow 2, \dots, H \rightarrow 8$. # depth=0

Step 3: Randomly organize the created matrix (goal matrix)
random matrix = (current state matrix)

Step 4: Run a loop

- Each subsequent possible block is checked & the cost is calculated.
- Cost is the number of blocks each block is misplaced from its current position to correct position.
- The final = cost + depth.
- Best final is stored for all possible next moves.
- The next move is predicted on the basis of the lowest final.
- To make each move such that the final is reduced.
- Increment the depth after each increment in level (loop).
- After every move check if winning condition is satisfied.

Step 5: When winning condition is satisfied display final & depth.

Step 6: Stop

Date _____
Page _____

Hill Climbing

Final = cost + depth
 $f(n) = h(n) + g(n)$

Current

1	2	3
4	7	5
6	8	□

depth = 0
cost = 2
final = 2

Goal

1	2	3
4	□	5
6	7	8

Final state

depth = 1

Moves:

- Up: $1+1=2$
- Right: $1+1=2$
- Left: $1+1=2$
- Down: $1+1=2$

Final state

depth = 2

Moves:

- Up: $1+2=3$
- Right: $1+2=3$
- Left: $1+2=3$
- Down: $1+2=3$

Final state

Non-blocked

Final = cost + depth
 $f(n) = h(n) + g(n)$

Goal

1	2	3
4	7	5
6	8	□

depth = 0
cost = 2
final = 2

depth = 1

Moves:

- Up: $1+1=2$
- Right: $1+1=2$
- Left: $1+1=2$
- Down: $1+1=2$

Final state

depth = 2

Moves:

- Up: $2+2=4$
- Right: $2+2=4$
- Left: $2+2=4$
- Down: $2+2=4$

Final state

Final state

Final state

Code:

```
import heapq

class PuzzleState:
    def __init__(self, board, parent=None, move="", depth=0, cost=0):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = cost # g(n) + h(n)

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

```

def find_blank(self):
    return self.board.index(0)

def get_moves(self):
    """Generate possible moves"""
    blank = self.find_blank()
    moves = []
    row, col = divmod(blank, 3)
    directions = {
        "Up": (row - 1, col),
        "Down": (row + 1, col),
        "Left": (row, col - 1),
        "Right": (row, col + 1)
    }
    for move, (r, c) in directions.items():
        if 0 <= r < 3 and 0 <= c < 3:
            new_blank = r * 3 + c
            new_board = self.board[:]
            new_board[blank], new_board[new_blank] = new_board[new_blank], new_board[blank]
            moves.append(PuzzleState(new_board, self, move, self.depth + 1))
    return moves

def path(self):
    """Reconstruct solution path"""
    node, p = self, []
    while node:
        p.append((node.move, node.board))
        node = node.parent
    return list(reversed(p))

def print_board(self):
    """Print the board in 3x3 matrix form"""
    for i in range(0, 9, 3):
        print(self.board[i:i+3]) # Print rows of the board

# ----- Heuristics -----
def misplaced_tiles(state, goal):
    return sum(1 for i in range(9) if state.board[i] != 0 and state.board[i] != goal[i])

def manhattan_distance(state, goal):
    dist = 0
    for i in range(9):
        if state.board[i] != 0:
            r1, c1 = divmod(i, 3)
            r2, c2 = divmod(goal.index(state.board[i]), 3)
            dist += abs(r1 - r2) + abs(c1 - c2)

```

```

return dist

# ----- A* Search -----
def a_star(start, goal, heuristic):
    open_list = []
    closed_set = set()
    start_state = PuzzleState(start)
    start_state.cost = heuristic(start_state, goal)
    heapq.heappush(open_list, start_state)

    while open_list:
        current = heapq.heappop(open_list)

        if current.board == goal:
            return current.path()

        closed_set.add(tuple(current.board))

        for neighbor in current.get_moves():
            if tuple(neighbor.board) in closed_set:
                continue
            neighbor.cost = neighbor.depth + heuristic(neighbor, goal)
            heapq.heappush(open_list, neighbor)

    return None

# ----- Example Run -----
# if __name__ == "__main__":
    initial_state = [1, 2, 3,
                    4, 7, 5,
                    6, 8, 0]

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

    print("A* with Misplaced Tiles:")
    solution1 = a_star(initial_state, goal_state, misplaced_tiles)
    for move, state in solution1:
        print(f"Move: {move}")
        PuzzleState(state).print_board() # Print the board as a matrix

    print("\nA* with Manhattan Distance:")
    solution2 = a_star(initial_state, goal_state, manhattan_distance)
    for move, state in solution2:

```

```

print(f"Move: {move}")
PuzzleState(state).print_board() # Print the board as a matrix
if __name__ == "__main__":
    initial_state = [1, 2, 3,
                     4, 7, 5,
                     6, 8, 0]

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

    print("A* with Misplaced Tiles:")
    solution1 = a_star(initial_state, goal_state, misplaced_tiles)
    for idx, (move, state) in enumerate(solution1):
        ps = PuzzleState(state)
        h = misplaced_tiles(ps, goal_state)
        g = idx # depth = move number in path
        cost = g + h
        print(f"Move: {move}, Depth: {g}, Heuristic: {h}, Total cost: {cost}")
        ps.print_board()

    print("\nA* with Manhattan Distance:")
    solution2 = a_star(initial_state, goal_state, manhattan_distance)
    for idx, (move, state) in enumerate(solution2):
        ps = PuzzleState(state)
        h = manhattan_distance(ps, goal_state)
        g = idx # depth
        cost = g + h
        print(f"Move: {move}, Depth: {g}, Heuristic: {h}, Total cost: {cost}")
        ps.print_board()

```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

<p>Lab 4 Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.</p> <p>Lab 5a Implement Hill Climbing Algorithms for N-Queens.</p> <p>Algorithm:-</p> <p>Step 1: START.</p> <p>Step 2: Initialization Generate a random initial state (currentState). Compute the objective value: $CurrentObjective = Objective(currentState) + conflicts(currentState)$</p> <p>Step 3: Generate Neighbors From the currentState generate all neighbors by moving queen attacking to a diff row within its col).</p> <p>Step 4: Evaluate Neighbors For each neighbor, compute its objective value: $Objective(neighbor) = conflicts(neighbor)$</p> <p>Step 5: Check Termination If currentState is local optimum $Objective(currentState) = 0$, go to step 6. If $Objective(currentState) > 0$, go to step 3.</p> <p>Step 6: Output</p>	<p>Conflict = Total pairs of queen attacking each other Objective Value = No. of conflicts</p> <p>Step 6: Output: When a solution with zero conflicts is found, output the current state as the solution</p> <p>Step 7: STOP.</p> <p>Output:-</p> <p>Initial Board: $\begin{matrix} \cdot & \cdot & Q & \cdot \\ Q & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix}$</p> <p>Initial conflict: 3</p> <p>Iteration 1: add row 1 to 3 conflict 1</p> <p>Iteration 2: col 3 row 0 to 2 conflict 0</p> <p>Iteration 3: Optimum conflict 0</p> <p>Final board: $\begin{matrix} \cdot & \cdot & Q & \cdot \\ Q & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix}$</p> <p>Total Conflict: 0</p> <p>Total Iteration: 3</p>	<p>Scoring</p> <p>Initial $\begin{matrix} \cdot & \cdot & Q & \cdot \\ Q & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix}$ conflict: 3</p> <p>Iteration 1 $\begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & Q \\ \cdot & \cdot & Q & \cdot \\ Q & \cdot & \cdot & \cdot \end{matrix}$ conflict: 2</p> <p>Iteration 2 $\begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & Q \\ Q & \cdot & \cdot & \cdot \end{matrix}$ conflict: 1</p> <p>Iteration 3 $\begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ Q & \cdot & \cdot & \cdot \end{matrix}$ conflict: 0</p> <p>Final $\begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ Q & \cdot & \cdot & \cdot \end{matrix}$ conflict: 0</p>
--	---	---

Code:

```
import random
```

```
def conflicts(state):
    """Count number of pairs of queens attacking each other."""
    n = len(state)
    count = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                count += 1
    return count
```

```
def generate_neighbors(state):
    """Generate all neighbors by moving one queen to another row in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if row != state[col]:
                neighbor = state.copy()
                neighbor[col] = row
                neighbors.append(neighbor)
```

```

neighbor[col] = row
neighbors.append(neighbor)
return neighbors

def print_board(state):
    """Print the board configuration."""
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            if state[col] == row:
                line += " Q "
            else:
                line += ". "
        print(line)
    print("\n")

def hill_climbing_n_queens(n, max_iterations=1000, max_restarts=50):
    def random_state():
        return [random.randint(0, n - 1) for _ in range(n)]

    current_state = random_state()
    current_conflicts = conflicts(current_state)

    print("Initial board:")
    print_board(current_state)
    print(f"Initial conflicts: {current_conflicts}\n")

    restarts = 0
    iteration = 0

    while restarts < max_restarts:
        improved = True
        while improved and iteration < max_iterations:
            iteration += 1
            neighbors = generate_neighbors(current_state)
            neighbor_conflicts = [conflicts(neigh) for neigh in neighbors]

            min_conflicts = min(neighbor_conflicts)
            min_indices = [i for i, val in enumerate(neighbor_conflicts) if val == min_conflicts]

            if min_conflicts < current_conflicts:
                chosen_index = random.choice(min_indices)
                next_state = neighbors[chosen_index]
                # Find which column changed and rows before/after
                for col in range(n):
                    if current_state[col] != next_state[col]:

```

```

        moved_col = col
        old_row = current_state[col]
        new_row = next_state[col]
        break
    direction = "up" if new_row < old_row else "down"
    print(f"Iteration {iteration}: Moved queen in column {moved_col} from row {old_row}"
{direction} to row {new_row}, conflicts {min_conflicts}")
    current_state = next_state
    current_conflicts = min_conflicts

elif min_conflicts == current_conflicts:
    chosen_index = random.choice(min_indices)
    next_state = neighbors[chosen_index]
    # Find which column changed and rows before/after
    for col in range(n):
        if current_state[col] != next_state[col]:
            moved_col = col
            old_row = current_state[col]
            new_row = next_state[col]
            break
    direction = "up" if new_row < old_row else "down"
    print(f"Iteration {iteration}: Moved queen in column {moved_col} from row {old_row}"
{direction} to row {new_row} (equal objective), conflicts {min_conflicts}")
    current_state = next_state
    current_conflicts = min_conflicts

else:
    print(f"Iteration {iteration}: Local optimum reached with conflicts {current_conflicts}")
    improved = False

if current_conflicts == 0:
    print("\nGlobal optimum found!")
    break
restarts += 1
current_state = random_state()
current_conflicts = conflicts(current_state)
print(f"\nRestart {restarts}: New random state with conflicts {current_conflicts}")
print("\nFinal board:")
print_board(current_state)
print(f"Final conflicts: {current_conflicts}")
print(f"Total iterations: {iteration}")
print(f"Total restarts: {restarts}")
return current_state
# Example: Solve 8-Queens
n = 5
hill_climbing_n_queens(n)

```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

Solve 8-puzzle (Simulated Annealing)

Step 1: start.

Step 2: Create a matrix (Ex: 3x3) Assign the first 3 elements as A, B, ..., M respectively.
Map each number of to a alphabets
 $A \rightarrow 1, B \rightarrow 2, \dots, M \rightarrow 8$.

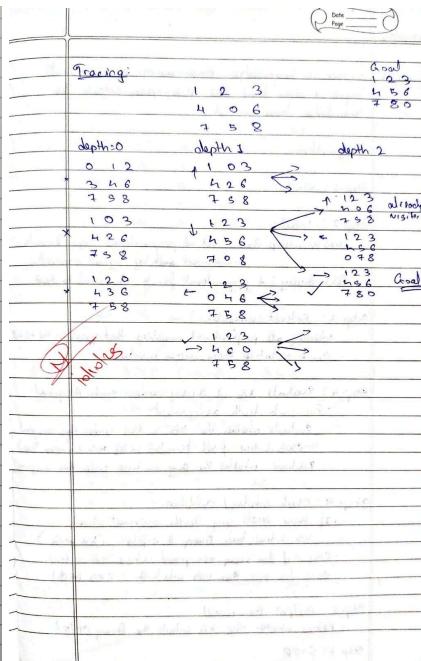
Step 3: Randomly organize the created matrix \rightarrow (goal matrix)
random matrix = (current state matrix)

Step 4: Run a loop.

- Each block is checked & their cost is calculated.
- Cost is the ~~number~~ of that blocks current position from the correct position. (cost=misplaced)
- Cost of all blocks are summed & stored (misplaced).
- The next move is predicted on the basis of the lowest sum i.e. on moving which block does the ~~costliest~~ heuristic is reduced.
- So make each move such that the cost is minimum.
- Log every move.
- After every move check if winning condition is satisfied.

Step 5: When winning condition is satisfied display as win & hours

Step 6: STOP.



Code:

```
// (8-Puzzle)
from collections import deque
```

```
class PuzzleState:
    def __init__(self, board, parent=None, move="", depth=0):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth # number of moves from start

    def find_blank(self):
        return self.board.index(0)

    def get_moves(self):
        """Generate possible moves from current state"""
        blank = self.find_blank()
        moves = []
        row, col = divmod(blank, 3)
        directions = {
            "Up": (row - 1, col),
            "Down": (row + 1, col),
            "Left": (row, col - 1),
```

```

    "Right": (row, col + 1)
}
for move, (r, c) in directions.items():
    if 0 <= r < 3 and 0 <= c < 3:
        new_blank = r * 3 + c
        new_board = self.board[:]
        new_board[blank], new_board[new_blank] = new_board[new_blank], new_board[blank]
        moves.append(PuzzleState(new_board, self, move, self.depth + 1))
return moves

def path(self):
    """Reconstruct the path from start to current state"""
    node, p = self, []
    while node:
        p.append((node.move, node.board))
        node = node.parent
    return list(reversed(p))

def print_board(self):
    """Print the board in 3x3 format"""
    for i in range(0, 9, 3):
        print(self.board[i:i+3])
    print()

def bfs(start, goal):
    start_state = PuzzleState(start)
    if start == goal:
        return start_state.path()

    queue = deque([start_state])
    visited = set()
    visited.add(tuple(start))

    while queue:
        current = queue.popleft()

        if current.board == goal:
            return current.path()

        for neighbor in current.get_moves():
            t_board = tuple(neighbor.board)
            if t_board not in visited:
                visited.add(t_board)
                queue.append(neighbor)
    return None

```

```

if __name__ == "__main__":
    initial_state = [
        1, 2, 3,
        4, 7, 5,
        6, 8, 0
    ]

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

    solution = bfs(initial_state, goal_state)

    if solution:
        print(f"Solution found in {len(solution)-1} moves:\n")
        for idx, (move, board) in enumerate(solution):
            print(f"Step {idx}: Move: {move}")
            for i in range(0, 9, 3):
                print(board[i:i+3])
            print()
    else:
        print("No solution found.")

```

Program 6

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

Algorithm:

Notes	
Lab 7	Create a knowledge base using propositional logic & show that the given query entails the knowledge base or not.
Algorithm:	
Step 1: Start	
Step 2: Input:	<p>(KB) Knowledge Base : set of propositional logic formulas combined by logical connectives</p> <p>(Q) Query : A propositional formula you want to test</p>
Step 3: Extract variables	<p>Identify all propositional variables that appear in the KB.</p> <p>Create a list of these unique variables.</p>
Step 4: Evaluate KB & Query under each assignment	<ul style="list-style-type: none"> For each truth assignment Evaluate whether the KB is true under this assignment KB is true if all formulas in the KB evaluate to true Evaluate whether the query is true under this assignment
Step 5: Check entailment condition:	<ul style="list-style-type: none"> If there exists any truth assignment where: <ul style="list-style-type: none"> KB is true but, Query Q is false ($KB \models Q$) else if for every assignment where KB is true, Q is also true, then KB entails Q ($KB \vdash Q$)
Step 6: Output the result.	<p>Return whether the KB entails the Query M/F</p>
Step 7: STOP	

Notes																																																							
Ans)	Query (α) = $A \vee B$																																																						
	$KB = (A \vee C) \wedge (\neg A \vee \neg C)$																																																						
	<table border="1"> <thead> <tr> <th>A</th><th>B</th><th>C</th><th>(α)</th><th>KB</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	C	(α)	KB	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	1	1	0	1	1	1	1	1	1	1									
A	B	C	(α)	KB																																																			
0	0	0	0	0																																																			
0	0	1	0	0																																																			
0	1	0	1	0																																																			
0	1	1	1	1																																																			
1	0	0	1	1																																																			
1	0	1	1	0																																																			
1	1	0	1	1																																																			
1	1	1	1	1																																																			
	$KB \models \alpha$ as ($\text{when } KB \rightarrow 1 \quad \alpha \rightarrow 1$)																																																						
	TOP																																																						
Ans)	$KB = Q \rightarrow P \quad P \rightarrow R \quad Q \vee R$ Does KB entail R? KB entails $R \rightarrow P$ KB entails $Q \rightarrow R$ $P \quad Q \quad R \quad KB \quad R \rightarrow P \quad Q \rightarrow R$ <table border="1"> <thead> <tr> <th>P</th><th>Q</th><th>R</th><th>KB</th><th>$R \rightarrow P$</th><th>$Q \rightarrow R$</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> </tbody> </table>	P	Q	R	KB	$R \rightarrow P$	$Q \rightarrow R$	0	0	0	0	1	1	0	0	1	1	1	1	0	1	0	0	1	0	0	1	1	0	0	1	1	0	0	0	1	1	1	0	1	1	1	1	1	1	0	0	1	0	1	1	1	0	1	1
P	Q	R	KB	$R \rightarrow P$	$Q \rightarrow R$																																																		
0	0	0	0	1	1																																																		
0	0	1	1	1	1																																																		
0	1	0	0	1	0																																																		
0	1	1	0	0	1																																																		
1	0	0	0	1	1																																																		
1	0	1	1	1	1																																																		
1	1	0	0	1	0																																																		
1	1	1	0	1	1																																																		
	So This																																																						

Code:

```
import itertools
import re
```

```
def extract_variables(expressions):
    vars = set()
    pattern = r'\b[A-Z]\b'
    for expr in expressions:
        vars.update(re.findall(pattern, expr))
    return sorted(vars)
```

```
def eval_expr(expr, assignment):
    for var, val in assignment.items():
        expr = re.sub(r'\b' + var + r'\b', str(val), expr)
    expr = expr.replace('¬', ' not ')

```

```

expr = expr.replace('∧', ' and ')
expr = expr.replace('∨', ' or ')

while '→' in expr:
    match = re.search(r'([^\(\)]+)*→*([^\(\)]+)', expr)
    if not match:
        expr = expr.replace('→', ' <= ')
        break
    left = match.group(1)
    right = match.group(2)
    replacement = f'((not ({left})) or ({right}))'
    expr = expr[:match.start()] + replacement + expr[match.end():]

try:
    return eval(expr)
except Exception as e:
    raise ValueError(f"Error evaluating expression '{expr}': {e}")

def entails(KB, query):
    all_exprs = KB + [query]
    variables = extract_variables(all_exprs)

    truth_table = []

    # Generate all possible truth assignments
    for values in itertools.product([False, True], repeat=len(variables)):
        assignment = dict(zip(variables, values))

        KB_true = all(eval_expr(kb, assignment) for kb in KB)
        query_true = eval_expr(query, assignment)

        truth_table.append((assignment, KB_true, query_true))

    if KB_true and not query_true:
        return False, truth_table

    return True, truth_table

# Hardcoded KB and query as per your example
if __name__ == "__main__":
    # Define KB and Query for the problem
    KB = ['(A ∨ C)', '(B ∨ ¬C)'] # Knowledge Base (KB)
    query = '(A ∨ B)' # Query (α)

    # Check entailment and get truth table

```

```
entails_result, truth_table = entails(KB, query)

# Output entailment result
print(f"Does KB entail '{query}'? {entails_result}")

# Print the truth table
print("\nTruth Table:")
header = ["Assignment", "KB Evaluation", "Query Evaluation"]
print(f'{header[0]}:{<25} {header[1]}:{<20} {header[2]}:{<20}')

for assignment, KB_true, query_true in truth_table:
    print(f'{str(assignment)}:{<25} {str(KB_true)}:{<20} {str(query_true)}:{<20}")
```

Program 7

Implement unification in first order logic

Algorithm:

Lab 9	Implement unification in first order logic.
	Algorithm:-
Step 1: Initialize	$\theta = \emptyset$ (empty) & the pair list $E = \{(\alpha, \beta)\}$.
Step 2: While E is not empty	Take one pair (α, β) from E .
Step 3: Apply current substitution	Apply θ to both terms: $\alpha = s\theta$ & $\beta = t\theta$.
Step 4: Compare	If $(s \neq t)$ are identical continue. Else if $(s \text{ is var})$ if $(s \text{ occurs in } t)$ FAIL \rightarrow (occurred). Else external substitut $\theta = \theta \cup (s \rightarrow t)$. Else if $(t \text{ is var})$ if $(t \text{ occurs in } s)$ FAIL \rightarrow (occurred). Else external subst $\theta = \theta \cup (t \rightarrow s)$. Else if $(\text{both functions or const})$ Add their org pair wise to E . Else FAIL

Date _____ Page _____
Step 5: Apply Substitution (whenever a new substitution $(x \rightarrow t)$ is added, apply it to all pairs and to θ).
Step 6: Return θ
Output: $P(a, x, f(g(y))) \neq P(x, f(z), f(u))$
1. $\theta = \emptyset$ 2. $a = z, x = f(z), g(y) = u$, 3. $\theta \cup \{z \rightarrow a, x \rightarrow f(a), f(g(y)) \rightarrow f(u), u \rightarrow g(y)\}$
$\theta = \{z \rightarrow a, x \rightarrow f(a), u \rightarrow g(y)\}$
Code output: fails \rightarrow (occurred)
Code output: exp 1 = ('P', '1', '2', 'x', 'C', ['(' 'a' ', 'f' 'g' 'y' ')']) exp 2 = ('P', '1', '2', 'x', 'C', 'f' 'u')]) match: { 'a': '2', 'x': '1', 'f': '1', 'g': '1' }

Code:

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

```
def occurs_check(var, expr):
    if var == expr:
        return True
    elif isinstance(expr, tuple):
        return any(occurs_check(var, sub) for sub in expr[1])
    return False
```

```
def unify(x, y, theta=None):
```

```
    if theta is None:
```

```
        theta = {}
```

```

if x == y:
    return theta
elif is_variable(x):
    if occurs_check(x, y):
        return None
    theta[x] = y
    return {k: substitute(v, theta) for k, v in theta.items()}
elif is_variable(y):
    return unify(y, x, theta)
elif isinstance(x, tuple) and isinstance(y, tuple) and x[0] == y[0]:
    for a, b in zip(x[1], y[1]):
        theta = unify(substitute(a, theta), substitute(b, theta), theta)
        if theta is None:
            return None
    return theta
else:
    return None

def substitute(expr, theta):
    if isinstance(expr, str):
        return theta.get(expr, expr)
    elif isinstance(expr, tuple):
        return (expr[0], [substitute(arg, theta) for arg in expr[1]])
    return expr

# ----- Example -----
# Represent P(a, x, f(g(y))) as ('P', ['a', 'x', ('f', [('g', ['y'])])])
# Represent P(z, f(z), f(u)) as ('P', ['z', ('f', ['z']), ('f', ['u'])])

expr1 = ('P', ['a', 'x', ('f', [('g', ['y'])])])
expr2 = ('P', ['z', ('f', ['z']), ('f', ['u'])])

result = unify(expr1, expr2)
print("Most General Unifier (MGU):", result)

```

Program 8

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

Algorithm:

31/10/25	<p>Lab-11 Create a knowledge base consisting of first order logic statements & prove the given query using forward reasoning</p> <p>Algorithm:</p> <p>Step 1: Initialize the Knowledge Base (KB)</p> <ul style="list-style-type: none"> Starts all the known facts in a set F. <p>Step 2:</p> <ul style="list-style-type: none"> For each rule in the KB, represent it as: $(P_1 \wedge P_2 \wedge \dots \wedge P_n) \rightarrow Q$ <p>Step 3: Repeat until no new facts</p> <ul style="list-style-type: none"> For each rule in KB <ul style="list-style-type: none"> If all premises P_1, P_2, \dots, P_n are in the set of known facts F then add the conclusion Q. Add Q to F if it is new <p>Step 4:</p> <ul style="list-style-type: none"> Whenever a new fact is inferred, print it. <p>Step 5:</p> <ul style="list-style-type: none"> After no new facts can be added, check whether the query exist in the set of derived facts F <p>Step 6:</p> <ul style="list-style-type: none"> If the query is proved \rightarrow True else \rightarrow False
----------	---

<p>Date _____ Page _____</p> <p>Output:</p> <p>Facts: Human(Socrates)</p> <p>Rules: Human(x) \rightarrow Mortal(x) Mortal(x) \rightarrow Dies(x)</p> <p>Query: Mortal(Socrates)</p> <p>1. Facts: A Human(Socrates) ? 2. Human(Socrates) \rightarrow Mortal Facts (Human(Socrates)), Mortal(Socrates) Mortal(\rightarrow) \rightarrow Dies(\rightarrow) Facts (Human(Socrates)), Mortal(Socrates), Dies(\rightarrow) ? 3. @Mortal(Socrates) is present in Facts Query is proved.</p> <p>In code:</p> <p>facts = set(["Human(Socrates)"]) rules = [("Human(x)", "Mortal(x)"), ("Mortal(x)", "Dies(x)")] query = "Mortal(Socrates)" result = True (Proved)</p>

Code:

```
from collections import defaultdict
```

```
# ----- Step 1: Define Knowledge Base -----
```

```
facts = set(["Human(Socrates)"])
```

```
# Rules are stored as (premises, conclusion)
```

```
rules = [  
        ("Human(x)", "Mortal(x)"),  
        ("Mortal(x)", "Dies(x)")  
    ]
```

```

query = "Mortal(Socrates)" # What we want to prove

# ----- Step 2: Substitute variables -----
def substitute(expr, var, val):
    return expr.replace(var, val)

# ----- Step 3: Forward Chaining Function -----
def forward_chain(facts, rules, query):
    new_facts = set(facts)
    added = True

    while added:
        added = False
        for premises, conclusion in rules:
            for fact in list(new_facts):
                # Check if any variable substitution is possible
                if '(' in fact:
                    const = fact[fact.find('(')+1:fact.find(')')]
                    temp = conclusion
                    for p in premises:
                        temp = substitute(temp, 'x', const)

                    if all(substitute(p, 'x', const) in new_facts for p in premises):
                        if temp not in new_facts:
                            print(f'Inferred: {temp}')
                            new_facts.add(temp)
                            added = True

    return query in new_facts

# ----- Step 4: Run the reasoning -----
print("Initial Facts:", facts)
result = forward_chain(facts, rules, query)
print("\nQuery:", query)
print("Result:", "PROVED" if result else "NOT PROVED")

```

Program 9

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

Algorithm:

<p>7/11/25</p> <p>Lab-8 Create a knowledge base using propositional logic & prove the given query using resolution</p> <p>Algorithm:</p> <p>Step 1: Input the Knowledge Base (KB) & Query.</p> <ul style="list-style-type: none"> Represent each FOL statement in CNF form. (a set of clauses, each a v of literals) Represent the query (Δ) to be proven. <p>Step 2: Negate the query & Add to the KB</p> <ul style="list-style-type: none"> To prove $\Delta \rightarrow \neg\Delta$ add it to knowledge base The goal is to derive a contradiction (empty clause) <p>Step 3: Standardize Variables & Prepare for Resolution</p> <ul style="list-style-type: none"> Ensure each clause has unique vars to avoid naming conflicts Apply unification to find var substitutions that make literals identical. <p>Step 4: Perform Resolution</p> <ul style="list-style-type: none"> Repeatedly pick two clauses ($C_i \wedge C_j$) that contain complementary literals (e.g., $P(x)$ and $\neg P(y)$). Apply unification to make them identical, then combine the remaining literals to form a new resolvent. 	<p>Date _____ Page _____</p> <p>Step 5: Check for contradiction</p> <ul style="list-style-type: none"> If an empty clause (\emptyset) is derived, a contradiction has been found. (True) Else no new clauses can be generated, the query can't be proven from the KB (False) <p>Step 6: Output the Result</p> <p>print proved or not proved.</p> <p><u>Output:</u></p> <p>KB: All humans are mortal $\forall x \text{Human}(x) \rightarrow \text{Mortal}(x)$ $\text{Socrates is a human} \quad \text{Human(Socrates)}$</p> <p>Query: Mortal(Socrates)</p> <p>Conversion to CNF</p> <p>$\neg \text{Human}(x) \vee \text{Mortal}(x)$ Human(Socrates)</p> <p>Negated Query: $\neg \text{Mortal(Socrates)}$</p> <p>Resolution steps:</p> <p>Resolus (1) & (2) $\Rightarrow Socrates \Rightarrow \text{derive Mortal(Socrates)}$</p> <p>Resolus with (3)</p> <p>$\text{Mortal(Socrates)} \text{ and } \neg \text{Mortal(Socrates)} \Rightarrow \emptyset$ (True)</p> <p><u>Output:</u></p> <p>True</p> <p><u>Input:</u></p> <p>$\Gamma(\text{Human}, [x], \text{True}), (\text{Mortal}, [x], \text{False})$ $\Gamma(\text{Human}, [Socrates], \text{False})$ $\Gamma(\text{Mortal}, [Socrates], \text{True})$</p> <p>$\neg \text{Prove} : (\text{Mortal}, [Socrates], \text{False})$</p> <p><i>Good</i> // True</p>
--	---

Code:

```
import copy
```

```
# Each clause is a disjunction of literals
# Each literal is represented as a tuple: ('predicate', [args], is_negated)
# Example: ('Human', ['x'], False) means Human(x)
# Example: ('Mortal', ['x'], True) means ¬Mortal(x)
```

```
# Example KB:
# 1. ¬Human(x) ∨ Mortal(x) (All humans are mortal)
# 2. Human(Socrates)
# Query: Mortal(Socrates)
```

```

KB = [
    ['Human', ['x'], True], ('Mortal', ['x'], False)],
    ['Human', ['Socrates'], False]
]
query = ('Mortal', ['Socrates'], False)

# ----- Unification Function -----
def unify(x, y, theta=None):
    if theta is None:
        theta = {}
    if x == y:
        return theta
    elif isinstance(x, str) and x.islower(): # variable
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.islower(): # variable
        return unify_var(y, x, theta)
    elif isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        for xi, yi in zip(x, y):
            theta = unify(xi, yi, theta)
        if theta is None:
            return None
        return theta
    elif isinstance(x, tuple) and isinstance(y, tuple):
        return unify(list(x), list(y), theta)
    else:
        return None

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta

# ----- Apply Substitution -----
def substitute(clause, theta):
    new_clause = []
    for (pred, args, neg) in clause:
        new_args = [theta.get(a, a) for a in args]
        new_clause.append((pred, new_args, neg))
    return new_clause

# ----- Resolution -----
def resolve(ci, cj):
    resolvents = []

```

```

for (pi, args_i, neg_i) in ci:
    for (pj, args_j, neg_j) in cj:
        if pi == pj and neg_i != neg_j:
            theta = unify(args_i, args_j)
            if theta is not None:
                new_ci = [lit for lit in ci if lit != (pi, args_i, neg_i)]
                new_cj = [lit for lit in cj if lit != (pj, args_j, neg_j)]
                merged = new_ci + new_cj
                resolvent = substitute(merged, theta)
                # remove duplicates
                resolvent = [lit for lit in resolvent if lit not in []]
                resolvents.append(resolvent)
return resolvents

def resolution(KB, query):
    negated_query = [(query[0], query[1], not query[2])]
    clauses = KB + [negated_query]
    new = []
    print("\nInitial clauses:")
    for c in clauses:
        print(" ", c)

    while True:
        n = len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            for r in resolvents:
                if r == []:
                    print("\nDerived empty clause → Query proven")
                    return True
                new.append(r)
        if all(x in clauses for x in new):
            print("\nNo new clauses → Query cannot be proven")
            return False
        for c in new:
            if c not in clauses:
                clauses.append(c)
    if __name__ == "__main__":
        print("Proving:", query)
        result = resolution(KB, query)
        print("\nResult:", "Proved" if result else "Not Proved")

```

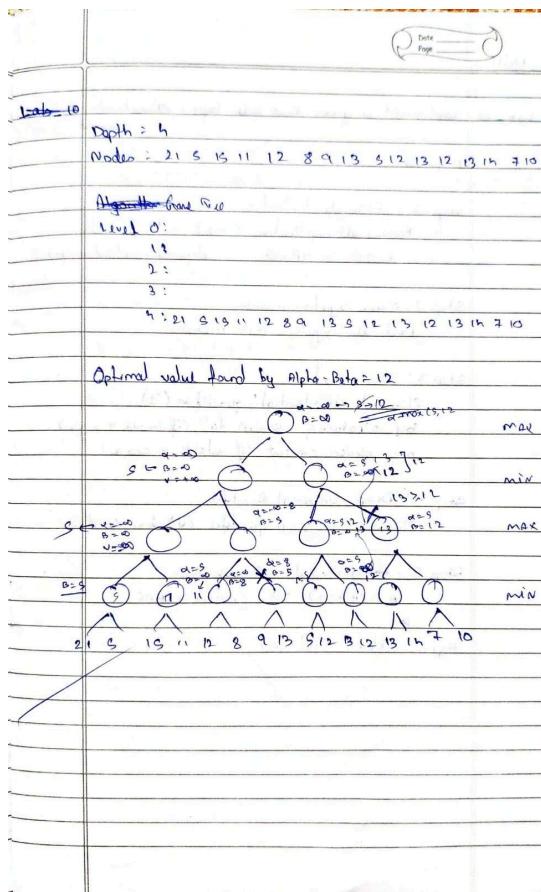
Program 10

Implement Alpha-Beta Pruning.

Algorithm:

Lab-12 Implement Alpha-Beta Pruning
Algorithm:
Step 1: Input. Root $d \leftarrow$ given depth $n = d^d$ (Leaf nodes)
Step 2: Build the game tree - construct a full binary tree (bottom-up). (internal nodes have two children, leaf nodes have N values)
Step 3: Define Variables - $\alpha = -\infty$ (Best value that MAX can guarantee so far) - $\beta = +\infty$ (MIN) TRUE \rightarrow MAX's turn FALSE \rightarrow MIN's turn
Step 4: Start Alpha-Beta Processing call(root, d, α , β , MAX) MAX node: value = $-\infty$ child.v = call(child, depth-1, α , β , MIN) value = max(value, child.v) $\alpha = \max(\alpha, value)$ if ($\alpha > \beta$) return (pruned) else return value

Date _____ Page _____
MIN Node: value = $+\infty$ child.v = call(child, depth-1, α , β , MAX) value = min(value, child.v) $\beta = \min(\beta, value)$ if ($\beta \leq \alpha$) return (pruned) else return value
Step 5: Terminate Condition. if the node is leaf or depth=0 return. Return the propagated value up to the root following minmax rules.
Step 6: Output. The value returned from the root is the optimal minmax value
Code: Exclusion
Depth: 3 Leaf Nodes: 10 9 11 18 5 3 50 3 Game Tree Level 0: [] 1: [] [] [] 2: [] [] [] [] 3: 10 9 11 18 5 3 50 3 Optimal value found by Alpha-Beta: <u>10</u>



Code:

```

import math

# -----
# Pretty Print Tree (Level Order)
# -----
def print_tree(root):
    print("\n--- Game Tree (Level Order) ---")
    queue = [root]
    level = 0
    while queue:
        next_level = []
        print(f"Level {level}: ", end="")
        for node in queue:
            if isinstance(node, int):
                print(node, end=" ")
            else:
                print("[*]", end=" ")
                next_level.extend(node)
        print()
        queue = next_level
  
```

```

        level += 1
        print("-----\n")

# -----
# Alpha-Beta Pruning with Logging
# -----
node_id_counter = 1

def alphabeta(node, depth, alpha, beta, maximizing):
    global node_id_counter
    my_id = node_id_counter
    node_id_counter += 1

    # Terminal leaf
    if depth == 0 or isinstance(node, int):
        print(f"Leaf Node {my_id}: value={node}")
        return node

    if maximizing:
        value = -math.inf
        print(f"MAX Node {my_id}: depth={depth}, alpha={alpha}, beta={beta}")

        for child in node:
            child_value = alphabeta(child, depth - 1, alpha, beta, False)
            value = max(value, child_value)
            alpha = max(alpha, value)
            print(f" MAX Node {my_id} updated: value={value}, alpha={alpha}, beta={beta}")

        if alpha >= beta:
            print(f" >>> PRUNING at MAX Node {my_id}: alpha >= beta ({alpha} >= {beta})")
            break

        print(f"MAX Node {my_id} returns {value}")
        return value

    else:
        value = math.inf
        print(f"MIN Node {my_id}: depth={depth}, alpha={alpha}, beta={beta}")

        for child in node:
            child_value = alphabeta(child, depth - 1, alpha, beta, True)
            value = min(value, child_value)
            beta = min(beta, value)
            print(f" MIN Node {my_id} updated: value={value}, alpha={alpha}, beta={beta}")

        if beta <= alpha:

```

```

print(f" >>> PRUNING at MIN Node {my_id}: beta <= alpha ({beta} <= {alpha})")
break

print(f"MIN Node {my_id} returns {value}")
return value

# -----
# Build FULL BINARY TREE
# -----
def build_tree(values):
    """Build binary tree bottom-up from leaf list."""
    level = values
    while len(level) > 1:
        next_level = []
        for i in range(0, len(level), 2):
            next_level.append([level[i], level[i + 1]])
        level = next_level
    return level[0]

# -----
# MAIN PROGRAM
# -----
if __name__ == "__main__":
    depth = int(input("Enter depth of tree: "))

    leaf_count = 2 ** depth
    print(f"Enter {leaf_count} leaf node values:")

    leaves = list(map(int, input().split()))
    if len(leaves) != leaf_count:
        print("Error: incorrect number of leaf values.")
        exit()

    # Build tree
    tree = build_tree(leaves)

    # Print tree
    print_tree(tree)

    # Run Alpha-Beta
    print("\n--- Alpha-Beta Evaluation ---")
    best_value = alphabeta(tree, depth, -math.inf, math.inf, True)

    print("\nOptimal value found by Alpha-Beta:", best_value)

```