VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Tanmay (1BM23CS360)

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 **Tanmay (1BM23CS360),** 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.

Megha J Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	20-8-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4-10
2	28-8-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	11-18
3	3-9-2025	Implement A* search algorithm	19-23
4	10-9-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	24-27
5	17-9-2025	Simulated Annealing to Solve 8-Queens problem	28-30
6	24-9-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	31-33
7	8-10-2025	Implement unification in first order logic	34-37
8	15-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	38-42
9	29-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	43-46
10	12-11-2025	Implement Alpha-Beta Pruning.	47-51

Github Link:

https://github.com/tanmayy-lab/AI-LAB

Program 1

a)Implement Tic –Tac –Toe Game

b)Implement vacuum cleaner agent

Algorithm:

TIC TAC TOE

```
function minimax ( board, depth, isMaximise ) :
        if terminal (win/loss/draw)
            return winner ( human/system) score
        if isMaximise :
            bestscore = float('-inf')
            make a move
            score = minimax (board, depth+1, False)
            undo move
            bestscore = max( score, bestscore)
            return bestscore
        else :
            bestscore = float('inf')
            make move
            score = minimax (board, depth+1, True)
            undo move
            bestscore = min(score, bestscore)
            return bestscore

function bestmove (board) :
        bestscore = float('-inf')
        for each empty cell :
            make move
            score = minimax (board, 0, False)
            undo move
            if (score > bestscore) :
                bestscore = score
            move this cell
        return move
```

```
main()
    create 3x3 board
    while game not over :
        if human turn :
            Take input and place 'O'
        else :
            compute bestmove() and place 'X'
```

State Space

VACUUM CLEANER.

DATE: 20/8/25 PAGE:

```
inpu roomA_status(clean/dirty)
inpu roomB_status(clean/dirty)
inpu vacuum_pos(A/B)
while RA == Dirty or RB == Dirty:
        if vacuum in current room is dirty:
                Clean it
        else:
                move to the other room
prin "Both are clean"
```

proceed

Code:

```
a)import math

# Print the board def
print_board(board):
for row in board:
print(" | ".join(row))
        print("-" * 5)

# Check winner def
check_winner(board):
    # Rows     for row in board:         if
row[0] == row[1] == row[2] != " ":
            return row[0]     # Columns     for col in range(3):
if board[0][col] == board[1][col] == board[2][col] != " ":
            return board[0][col]     # Diagonals     if
board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]     if board[0][2] ==
board[1][1] == board[2][0] != " ":
        return board[0][2]
    return None

# Check if moves left def
is_moves_left(board):
for row in board:         if "
" in row:             return
True
    return False

# Minimax function def
minimax(board, depth, isMax):
winner = check_winner(board)
if winner == "X":  # Computer
return 10 - depth     if winner ==
"O":  # Human         return depth -
10     if not is_moves_left(board):
        return 0

    if isMax:  # Computer turn
best = -math.inf         for i in
range(3):
            for j in range(3):                if board[i][j] == " ":
board[i][j] = "X"                    best = max(best, minimax(board,
depth + 1, False))                    board[i][j] = " "         return best
```

```
else:  # Human turn         best = math.inf         for i in range(3):
for j in range(3):                if board[i][j] == " ":
board[i][j] = "O"                     best = min(best, minimax(board,
depth + 1, True))                     board[i][j] = " "
        return best

# Find best move for computer def
best_move(board):     bestVal = -math.inf
move = (-1, -1)     for i in range(3):         for j
in range(3):             if board[i][j] == " ":
board[i][j] = "X"                 moveVal =
minimax(board, 0, False)
                board[i][j] = " "
if moveVal > bestVal:
move = (i, j)
bestVal = moveVal
    return move

# Game loop def
play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    human = "O"
    computer = "X"

    print("Tic-Tac-Toe Game!")
    print_board(board)

    while True:         # Human move         row, col = map(int,
input("Enter row and col (0-2): ").split())         if board[row][col]
!= " ":             print("Cell already taken!")             continue
board[row][col] = human

        if check_winner(board) == human:
            print_board(board)
print("You win!")
            break         if not
is_moves_left(board):
            print_board(board)
print("It's a Draw!")
            break

        # Computer move
print("Computer is thinking...")         r,
c = best_move(board)
        board[r][c] = computer
```

```
        print_board(board)

        if check_winner(board) == computer:
            print("Computer wins!")
break        if not
is_moves_left(board):
            print("It's a Draw!")
            break


play_game()



b)class VacuumCleaner:     def __init__(self,
room_A, room_B, vacuum_pos):
        self.room_A = room_A
self.room_B = room_B
        self.vacuum_pos = vacuum_pos

    def show_status(self):        print(f"Vacuum in Room {self.vacuum_pos} | Room A:
{self.room_A}, Room B: {self.room_B}")

    def clean(self):
        print("Initial State:")
self.show_status()
        print("------")

        steps = 0        while self.room_A == "Dirty" or
self.room_B == "Dirty":            steps += 1
            print(f"Step {steps}:")

            if self.vacuum_pos == "A":                if
self.room_A == "Dirty":                    print("Room
A is Dirty → Cleaned!")                    self.room_A
= "Clean"                else:
                    print("Room A is Clean → Move to Room B")
                    self.vacuum_pos = "B"

            elif self.vacuum_pos == "B":                if
self.room_B == "Dirty":                    print("Room
B is Dirty → Cleaned!")                    self.room_B
= "Clean"                else:
                    print("Room B is Clean → Move to Room A")
```

```
            self.vacuum_pos = "A"

        self.show_status()
        print("------")

    print("Goal Reached! Both rooms are clean.")


# ---- Main Program ---- room_A = input("Enter status of Room A
(Clean/Dirty): ").capitalize() room_B = input("Enter status of Room B
(Clean/Dirty): ").capitalize() vacuum_pos = input("Enter initial position
of Vacuum (A/B): ").upper()

agent = VacuumCleaner(room_A, room_B, vacuum_pos) agent.clean()
```

Output: a)

```
Enter row and col (0-2): 0 0
Computer is thinking...
O |   |
-----
  | X |
-----
  |   |
-----
Enter row and col (0-2): 2 1
Computer is thinking...
O |   |
-----
X | X |
-----
  | O |
-----
Enter row and col (0-2): 2 2
Computer is thinking...
O |   |
-----
X | X | X
-----
  | O | O
-----
Computer wins!
```

b)

```
Enter status of Room A (Clean/Dirty): Dirty
Enter status of Room B (Clean/Dirty): Clean
Enter initial position of Vacuum (A/B): A
Initial State:
Vacuum in Room A | Room A: Dirty, Room B: Clean
------
Step 1:
Room A is Dirty → Cleaned!
Vacuum in Room A | Room A: Clean, Room B: Clean
------
Goal Reached! Both rooms are clean.
```

Program 2:

a)Implement 8 puzzle problems using Depth First Search (DFS)

b)Implement Iterative deepening search algorithm

Algorithm:

DATE: 3/9/25 PAGE:

8 PUZZLE PROBLEM

DFS :

```
DFS( start_state, goal_state ) :
    stack ← empty
    visited ← empty set
    push ( start_state, [ ] ) into stack
    while stack not empty :
        (state, path) ← pop(stack)
        if state = goal_state :
            return path
        if state not in visited :
            add state to visited .
        for each neighbor in expand (state):
            newpath ← path + [neighbor]
            push (neighbour, newpath) into stack
    return 'No soln'
```

Initial

1	2	3
	4	5
7	6	8

goal:

1		3
4	2	5
7	6	8

right, up, down

1	2	3
4		5
7	6	8

	2	3
1	4	5
7	6	8

1	2	3
7	4	5
	6	8

up, right, down.

goal ✓

1		3
4	2	5
7	6	8

1	2	3
4	5	
7	6	8

$l = 2$

IDS :

DATE:

PAGE:

```
IDS (start, goal) :
        depth = 0
        while True :
                result ← DLS (start, goal, depth)
                if result ≠ 'cutoff' :
                        return result
                depth = depth + 1
DLS (start, goal, limit) :
        return RecursiveDLS (start, goal, limit, [])
RecursiveDLS (start, goal, limit, path) :
        if state = goal :
                return path
        if limit = 0 :
                return 'cutoff'
        cutoff_occured = False
        for each neighbor in expand(state) :
                if neighbour not in path :
                        result ← RecursiveDLS (neighbor,
                                goal, limit-1,
                                path + [neighbor])
                if result = 'cutoff' :
                        cutoff_occured = True
                else if result ≠ 'Failure' :
                        return result.
        if cutoff_occured :
                return 'cutoff'
        else :
                return 'failure'.
```

3/9/25

Code:
a)
```
import collections

# Define the GOAL_STATE for the 8-puzzle.
# This serves as a concrete example of a "state" in a search problem.
GOAL_STATE = (
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 0)  # 0 represents the blank space
)

def find_blank(state):
    """
    Helper function to find the coordinates of the blank space (0) in a state.
    """     for r in range(3):
for c in range(3):
if state[r][c] == 0:
            return r, c
return None

def get_neighbors(state):
    """
    Generates all valid neighbor states from the current state.

    In a search problem, this function defines the "transitions" between states.
For the 8-puzzle, this means moving the blank tile.

    Args:        state (tuple): The current
puzzle state.

    Returns:
        list: A list of new, valid puzzle states (tuples).
    """
    r_blank, c_blank = find_blank(state)
neighbors = []

    # Possible moves for the blank space: (dr, dc) -> Up, Down, Left, Right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dr, dc in moves:
        new_r, new_c = r_blank + dr, c_blank + dc

        # Check if the new position is within the grid boundaries
if 0 <= new_r < 3 and 0 <= new_c < 3:
```

```
        # Create a mutable copy of the state to make a change
        new_state_list = [list(row) for row in state]

        # Swap the blank space with the tile at the new position
new_state_list[r_blank][c_blank], new_state_list[new_r][new_c] = \
new_state_list[new_r][new_c], new_state_list[r_blank][c_blank]

        # Convert the list back to a tuple for hashability
new_state_tuple = tuple(tuple(row) for row in new_state_list)
neighbors.append(new_state_tuple)

    return neighbors

def dfs_with_states(initial_state):
    """
    Performs a Depth-First Search on a state-space problem.

    This function explicitly uses a stack to manage states and a set to
track visited states, demonstrating the core concepts of DFS.

    Args:        initial_state (tuple): The starting configuration of
the puzzle.

    Returns:
        list: The path to the goal state if found, or None.
    """
    # The stack will store tuples of (state, path_to_state)
    stack = collections.deque([(initial_state, [initial_state])])

    # The set stores visited states to prevent cycles and redundant work.
    visited = {initial_state}

    while stack:
        current_state, path = stack.pop()

        # Check if the current state is the goal state
if current_state == GOAL_STATE:
        return path

        # Explore the neighbors of the current state
        # The order of neighbors determines the search path (depth-first).
for neighbor in get_neighbors(current_state):
        # If the neighbor has not been visited, add it to the stack
if neighbor not in visited:                visited.add(neighbor)
new_path = path + [neighbor]
```

```
            stack.append((neighbor, new_path))

    return None # No solution found

def print_path(path):
    """
    Prints the sequence of states in the solution path.
    """     if path is
None:
        print("No solution found.")
        return

    for i, state in enumerate(path):
print(f"--- Step {i} ---")        for
row in state:
            print(" ".join(map(str, row)))
    print(f"--- Goal Reached in {len(path)-1} steps ---")

# --- Example Usage --- if
__name__ == "__main__":
    initial_state = (
(1, 2, 3),
        (4, 5, 6),
        (7, 0, 8)
    )

    print("Solving the 8-puzzle using DFS with explicit states...")
solution_path = dfs_with_states(initial_state)

    if solution_path:
        print_path(solution_path)
else:
        print("No solution found.")
```

b)

```
import collections

# Define the goal state
GOAL_STATE = (
    (1, 2, 3),
    (4, 5, 6),
```

```
    (7, 8, 0)
)

def find_blank(state):
    """Finds the coordinates of the blank space."""
for r in range(3):        for c in range(3):            if
state[r][c] == 0:
                return r, c
return None

def get_neighbors(state):
    """Generates all valid neighbor states."""
r_blank, c_blank = find_blank(state)
neighbors = []     moves = [(-1, 0), (1, 0),
(0, -1), (0, 1)]     for dr, dc in moves:
        new_r, new_c = r_blank + dr, c_blank + dc
if 0 <= new_r < 3 and 0 <= new_c < 3:
            new_state_list = [list(row) for row in state]
new_state_list[r_blank][c_blank], new_state_list[new_r][new_c] = \
new_state_list[new_r][new_c], new_state_list[r_blank][c_blank]
new_state_tuple = tuple(tuple(row) for row in new_state_list)
neighbors.append(new_state_tuple)     return neighbors

def depth_limited_search(initial_state, limit):
    """
    The DLS function is a core component of IDS.
    """
    stack = collections.deque([(initial_state, [initial_state])])
visited = {initial_state}

    while stack:
        current_state, path = stack.pop()

        if current_state == GOAL_STATE:
            return path

        if len(path) > limit:
            continue

        for neighbor in get_neighbors(current_state):
if neighbor not in visited:
visited.add(neighbor)                new_path = path
+ [neighbor]
                stack.append((neighbor, new_path))
```

```
    return None

def iterative_deepening_search(initial_state):
    """
    Solves the 8-puzzle using Iterative Deepening Search (IDS).

    IDS performs a series of DLS searches, with the depth limit increasing
with each iteration.

    Args:        initial_state (tuple): The starting
configuration.

    Returns:
        list: The path to the goal state if a solution exists.
"""
    # The depth limit starts at 0 and increases with each iteration.
    depth = 0
    # A safe upper bound for the depth to prevent infinite loops on unsolvable puzzles
# For a 3x3 puzzle, the maximum path length is about 31 steps.     max_depth = 50

    while depth < max_depth:
        print(f"Trying depth limit: {depth}")        solution =
depth_limited_search(initial_state, limit=depth)
                if
solution:
            return solution

        # Increment the depth limit for the next iteration
depth += 1

    return None # No solution found within the maximum depth

def print_path(path):
    """Prints the sequence of states in the solution path."""
if path is None:
        print("No solution found.")
        return

    for i, state in enumerate(path):
        print(f"--- Step {i} ---")
for row in state:
            print(" ".join(map(str, row)))     print(f"---
Goal Reached in {len(path)-1} steps ---")
```

```
# --- Example Usage --- if
__name__ == "__main__":
    initial_state = (
(1, 2, 3),
        (4, 5, 6),
        (7, 0, 8)
    )

    print("Solving the 8-puzzle with IDS...")
    solution_path = iterative_deepening_search(initial_state)

    if solution_path:
        print_path(solution_path)
else:
        print("Could not find a solution.")
```

Output: a)

```
Solving the 8-puzzle using DFS with explicit states...
--- Step 0 ---
1 2 3
4 5 6
7 0 8
--- Step 1 ---
1 2 3
4 5 6
7 8 0
--- Goal Reached in 1 steps ---
```

b)

```
Solving the 8-puzzle with IDS...
Trying depth limit: 0
Trying depth limit: 1
--- Step 0 ---
1 2 3
4 5 6
7 0 8
--- Step 1 ---
1 2 3
4 5 6
7 8 0
--- Goal Reached in 1 steps ---
```

Program 3:

Implement A* search algorithm

Algorithm:

A* Heuristic - Misplaced tiles

A_star_Misplaced Tiles(start, goal):
    open = priority queue ordered by f(n)
    closed = empty set
    Insert start into open with f(start) = g(start) + h(start)

    While open is not empty:
        node = extract_min(OPEN)
        if node.state == goal:
            return reconstruct_path(node)
        add node.state to closed

        for each neighbour in expand(node):
            if neighbor.state in closed:
                continue
            g(neighbor) = g(node) + 1
            h(neighbor) = countMisplacedTiles
                (neighbor, state, goal)
            f(neighbor) = g(neighbor) +
                h(neighbor)

            if neighbor not in open or better
                f found:
                set parent(neighbor) = node
            insert/update neighbor in OPEN

A* Manhattan Distance

A_star_ManhattanDistance(start, goal):
    open = priority queue ordered by f(n)
    closed = empty set
    insert start into open with f(start) = g(start) + h(start)

    while open is not empty:
        node = extract_min(OPEN)
        if node.state == goal:
            return reconstruct_path(node)
        add node.state to closed

        for each neighbor in expand(node):
            if neighbor.state in closed:
                continue
            g(neighbor) = g(node) + 1
            h(neighbor) = sumManhattan
                distance(neighbor, state, goal)
            f(neighbor) = g(neighbor) + h(neighbor)

        If neighbor not in open or better f not found:
            set parent(neighbor) = node

        insert/update neighbor in open.

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

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

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

    def get_moves(self):
```

```
        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):
        node, p = self, []
while node:
            p.append((node.move, node.board, node.depth))
node = node.parent
        return list(reversed(p))


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


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


def print_solution(solution):
print("Solution steps:")     for move,
state, depth in solution:
        indent = "  " * depth  # indent based on depth for tree-like view
if move == "":
            print(f"{indent}Start State (Depth {depth}):")
else:
            print(f"{indent}Move {move} (Depth {depth}):")
for i in range(0, 9, 3):
            print(indent + " ".join(str(x) if x != 0 else " " for x in state[i:i+3]))
print()


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

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

    print("A* with Misplaced Tiles Heuristic:\n")     solution1
= a_star(initial_state, goal_state, misplaced_tiles)     if
solution1:
        print_solution(solution1)
else:
        print("No solution found.")

    print("\nA* with Manhattan Distance Heuristic:\n")
    solution2 = a_star(initial_state, goal_state, manhattan_distance)
    if solution2:
```

```
    print_solution(solution2)
else:
    print("No solution found.")
```

Output:

```
A* with Misplaced Tiles Heuristic:

Solution steps:
Start State (Depth 0):
1 2 3
4   6
7 5 8

 Move Down (Depth 1):
 1 2 3
 4 5 6
 7   8

  Move Right (Depth 2):
  1 2 3
  4 5 6
  7 8


A* with Manhattan Distance Heuristic:

Solution steps:
Start State (Depth 0):
1 2 3
4   6
7 5 8

 Move Down (Depth 1):
 1 2 3
 4 5 6
 7   8

  Move Right (Depth 2):
  1 2 3
  4 5 6
  7 8
```

Program 4:

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Week-4 DATE: PAGE:

Hill climbing for N Queens.

```
function HILL_CLIMBING (n) :
        state = RANDOM_STATE(n)
        while True :
            neighbour = BEST_NEIGHBOR (state)
            if Heuristic (neighbor) >= heuristic (state) :
                return state .
            state = neighbor .

function random_state (n) :
        return [random row for col in range(n)]

function heuristic (state) :
        conflicts = 0
        for each pair of queens (i, j) :
            if in same row or diagonal (state[i], state[j]) :
                conflicts += 1
        return -conflicts .

function best_neighbor (state) :
        best = state
        best_score = heuristic (state)
        for each queen q in state :
            for each possible new_row in board :
                new_state = move_queen (state, q,
                                        new_row)
                if heuristic (new_state) < best_score
                    best = new_state
                    best_score = heuristic (new
                                            state)
        return best .
```

State space tree :

Queen :

$x_0 = 3$ $x_1 = 1$ $x_2 = 2$ $x_3 = 0$

	0	1	2	3
0				x_3
1		x_1		
2			x_2	
3	x_0			

x_0	x_1	x_2	x_3
3	1	2	0
1	3	2	0
2	1	3	0
0	1	2	3
3	2	1	0
3	0	2	1
3	1	0	2

pairs	h(cost)
x_0x_1	1
x_0x_2	1
x_0x_3	6
x_1x_2	6
x_1x_3	1
x_2x_3	1

	1	3	2	0
0				x_3
1	x_0			
2			x_2	
3		x_1		

(x_1, x_2) cost = 1

	2	1	3	0
0				x_3
1		x_1		
2	x_0			
3			x_2	

x_0x_1 cost = 1

	0	1	2	3
0	x_0			
1		x_1		
2			x_2	
3				x_3

h(cost) = 6

	3	2	1	0
0				x_3
1			x_2	
2		x_1		
3	x_0			

h(cost) = 6.

	3	0	2	1
0		x_1		
1				x_3
2			x_2	
3	x_0			

h(cost) = 1

	3	1	0	2
0			x_2	
1		x_1		
2				x_3
3	x_0			

h(cost) = 1

Lead : x_0x_1 : (1 3 2 0)

Current :

1	3	2	0	pairs	h(cost)
3	1	2	0	x_0x_1	2
2	3	1	0	x_0x_2	2
0	3	2	1	x_0x_3	4
1	2	3	0	x_1x_2	4
1	0	2	3	x_1x_3	2
1	3	0	2	x_2x_3	0

	3	1	2	0
0				x_3
1		x_1		
2			x_0	
3	x_0			

cost = 2

	1	3	0	2
0			x_2	
1	x_0			
2				x_3
3		x_1		

cost = 0

	2	3	1	0
0				x_3
1			x_2	
2	x_0			
3		x_1		

f(cost) = 2

	0	3	2	1
0	x_0			
1				x_3
2			x_2	
3		x_1		

cost = 4

	1	2	3	0
0				x_3
1	x_0			
2		x_1		
3			x_2	

cost = 4

	1	0	2	3
0		x_1		
1	x_0			
2			x_2	
3				x_3

cost = 2

Code:

```
import random

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

def heuristic(state):

    h = 0     n = len(state)     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):
                h += 1
    return h

def get_neighbors(state):

    neighbors = []     n = len(state)
for row in range(n):         for col in
range(n):             if col != state[row]:
neighbor = state.copy()
neighbor[row] = col
neighbors.append(neighbor)     return
neighbors

def hill_climbing(initial_state):
current = initial_state
    current_h = heuristic(current)

    while True:
        neighbors = get_neighbors(current)         neighbor_h_pairs =
[(heuristic(neighbor), neighbor) for neighbor in neighbors]

        # Find neighbor with minimum heuristic
neighbor_h_pairs.sort(key=lambda x: x[0])
        best_h, best_neighbor = neighbor_h_pairs[0]

        # If no improvement, return current state
if best_h >= current_h:             return current,
current_h

        current = best_neighbor
```

```
        current_h = best_h

def solve_n_queens(n=4):

    initial_state = [random.randint(0, n-1) for _ in range(n)]
solution, h = hill_climbing(initial_state)

    if h == 0:
        print("Solution found:")
print_board(solution)     else:
        print("No solution found with initial state, try again.")
print("Final state with heuristic", h)         print_board(solution)

# Run the solver
solve_n_queens()
```

Output:

Program 5:

Simulated Annealing to Solve 8-Queens problem

Algorithm:

Week-5

Simulated Annealing

DATE: 20/9/25

PAGE:

```
function simulated_annealing () :
        state = random_state (8)
        T = initial_temperature
        while T > min_temp :
                neighbor = random_neighbor (state)
                ΔE = heuristic (neighbor) - heuristic(state)
                if ΔE < 0 :
                        state = neighbor
                else if random() < exp (-ΔE / T)
                        state = neighbor
                T = COOL(T)
        return T

function random_neighbor (state) :
        new_state = copy (state)
        col = random_column ()
        row = random_row ()
        newstate [col] = row
        return new_state

function cool (T) :
        return T * 0.95
```

Output :

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

Code:

```
import random
import math

def print_board(board):
    n = len(board)
for i in range(n):
        row = ["Q" if board[i] == j else "." for j in range(n)]
print(" ".join(row))
    print()

def calculate_cost(board):
    """Heuristic: number of pairs of queens attacking each other"""
n = len(board)     cost = 0     for i in range(n):         for j in range(i +
1, n):             if board[i] == board[j] or abs(board[i] - board[j]) ==
abs(i - j):
                cost += 1
    return cost

def random_neighbor(board):
    """Generate a random neighboring board by moving one queen"""
n = len(board)     neighbor = list(board)     row = random.randint(0,
n - 1)     col = random.randint(0, n - 1)     neighbor[row] = col
    return neighbor

def simulated_annealing(n, initial_temp=100, cooling_rate=0.95, stopping_temp=1):
    current_board = [random.randint(0, n - 1) for _ in range(n)]
current_cost = calculate_cost(current_board)     temperature =
initial_temp
    step = 1

    print("Initial Board:")
print_board(current_board)     print(f"Initial
Cost: {current_cost}\n")

    while temperature > stopping_temp and current_cost > 0:
        neighbor = random_neighbor(current_board)
neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        # Acceptance probability         if delta < 0 or random.random() <
math.exp(-delta / temperature):
            current_board = neighbor
            current_cost = neighbor_cost

        print(f"Step {step}: Temp={temperature:.3f}, Cost={current_cost}")
step += 1
        temperature *= cooling_rate
```

```
    print("\nFinal Board:")
print_board(current_board)
print(f"Final Cost: {current_cost}")     if
current_cost == 0:
        print("Goal State Reached!")
else:
        print("Terminated before reaching goal.")

# Run for 8-Queens
simulated_annealing(8)
```

Output:

```
Initial Board:
...Q....
...Q....
......Q.
......Q.
.......Q
......Q.
......Q.
......Q.

Initial Cost: 15
```

```
Final Board:
Q.......
...Q....
.....Q..
Q.......
..Q.....
Q.......
......Q.
.Q......

Final Cost: 4
Terminated before reaching goal.
```

Program 6:

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

Algorithm:

Propositional logic

Algorithm:

function TT-entails? (KB, α):
KB: knowledge base
α: query, sentence in propositional logic

symbol ← a list of symbols in KB and α
return TT-check-All (KB, α, symbols, {})

function TT-CHECK-ALL (KB, α, symbols, model)
if EMPTY? (symbols) then
if PL-TRUE? (KB, model) then return
PL-TRUE? (α, model)
else return true
else do
P ← FIRST (symbols)
rest ← REST (symbols)
return (TT-check-all (KB, α, rest, model ∪ {P = true}) and TT-check-all (KB, α, rest, model ∪ {P = false}))

Output: Does KB entail R? True
Does KB entail R⇒P? False
Does KB entail Q⇒R? True

KB: Q→P, P→¬Q, Q∨R, R→P, Q→R

P	Q	R	Q→R	P→¬Q	Q∨R	KB (conversion of all)
0	0	0	1	1	0	F
0	0	1	1	1	1	T
0	1	0	0	1	1	F
0	1	1	0	1	1	F
1	0	0	1	1	0	F
1	0	1	1	1	1	T
1	1	0	1	0	1	F
1	1	1	1	0	1	F

α:

R→P	Q→R
T	T
F	T
T	F
F	T
T	T
T	T
T	F
T	T

For α = R→P: When KB T, α must be T
KB ⊭ α (KB DOES NOT ENTAIL α)

For α = Q→R: When KB is T, α is T
KB ⊨ α (KB entails α)

KB entails R

Code:

```
from itertools import product

def extract_symbols(expr):
    symbols = set()
for c in expr:
if c.isalpha():
            symbols.add(c)
    return sorted(symbols)

def replace_implications(expr):
# If no implication, return as is
if '=>' not in expr:
        return expr
```

```
    # Split on first occurrence of =>
left, right = expr.split('=>', 1)     left =
left.strip()     right = right.strip()
    # Replace implication by ((not (left)) or (right))
new_expr = f"((not ({left})) or ({right}))"     return
new_expr

def eval_expr(expr, model):     for
sym, val in model.items():         expr =
expr.replace(sym, str(val))     expr =
expr.replace('~', ' not ')     expr =
expr.replace('&', ' and ')
    expr = expr.replace('|', ' or ')

    # Replace implication once
    expr = replace_implications(expr)

    expr = expr.replace('<=>', '==')

    return eval(expr)

def truth_table(KB_sentences):
    symbols = set()     for s in
KB_sentences:         symbols |=
set(extract_symbols(s))
    symbols = sorted(symbols)

    print("Truth Table:")     header =
symbols + KB_sentences     print(" |
".join(f"{h:8}" for h in header))
    print("-" * (10 * (len(header))))

    models_where_KB_true = []     for values in
product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        KB_vals = [str(eval_expr(sentence, model)) for sentence in KB_sentences]
row_vals = [str(model[s]) for s in symbols]         print(" | ".join(f"{v:8}" for v in
(row_vals + KB_vals)))         if all(val == 'True' for val in KB_vals):
models_where_KB_true.append(model)     print("\nModels where KB is true:")
for m in models_where_KB_true:
        print(m)
    return models_where_KB_true

def tt_entails(KB_sentences, query):
    symbols = set()     for s in
KB_sentences + [query]:         symbols
|= set(extract_symbols(s))
    symbols = sorted(symbols)
```

```
    for values in product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        if all(eval_expr(sentence, model) for sentence in KB_sentences):
if not eval_expr(query, model):
            return False
return True

# Your KB and queries
KB = [
    "Q => P",
    "P => ~Q",
    "Q | R"
]

queries = [
"R",
    "R => P",
    "Q => R"
]

models = truth_table(KB)

for q in queries:
    print(f"\nDoes KB entail '{q}'? {tt_entails(KB, q)}")
```

Output:

```
---------------- RESTART: C:/Users/student/Desktop/ai_week6.py ----------------
Truth Table:
P        | Q        | R        | Q => P   | P => ~Q  | Q | R
-----------------------------------------------------------
True     | True     | True     | True     | False    | True
True     | True     | False    | True     | False    | True
True     | False    | True     | True     | True     | True
True     | False    | False    | True     | True     | False
False    | True     | True     | False    | True     | True
False    | True     | False    | False    | True     | True
False    | False    | True     | True     | True     | True
False    | False    | False    | True     | True     | False

Models where KB is true:
{'P': True, 'Q': False, 'R': True}
{'P': False, 'Q': False, 'R': True}

Does KB entail 'R'? True

Does KB entail 'R => P'? False

Does KB entail 'Q => R'? True
```

Program 7:

Implement unification in first order logic

Algorithm:

DATE: 15/10/25 PAGE:

WEEK 7

Unification in FOL

Algorithm : Unify (ψ_1, ψ_2)

Step 1 : if ψ_1 or ψ_2 is variable or constant, then

(a) If ψ_1, ψ_2 are identical, return NIL.

(b) If ψ_1 is a variable,

a. then if ψ_1 occurs in ψ_2, then return FAILURE

b. Else return $\{\{\psi_2/\psi_1\}\}$.

(c) If ψ_2 is a variable,

a. If ψ_2 occurs in ψ_1, return failure.

b. Else return $\{\{\psi_1/\psi_2\}\}$.

(d) Else return FAILURE.

Step 2 : If initial predicate symbol in ψ_1 and ψ_2 are not same, then return FAILURE.

Step 3 : If ψ_1 and ψ_2 have different number of arguments, return failure.

Step 4 : Set substitution set (SUBST) to NIL.

Step 5 : for i=1 to number of elements in ψ_1.

(a) Call unify with ith element of ψ_1 and ith element of ψ_2, put result in S.

(b) if S = failure return FAILURE

(c) if S ≠ NIL do,

a. Apply S to remainder of L1 & L2

b. SUBST = append (S, SUBST)

Step 6 : Return SUBST

DATE: 15/10/25 PAGE:

First Order Logic - Unification

Ex : P(x,y) P(a,f(z))

{x/a}

P(a,y) P(a,f(z))

{y/f(z)}

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

Output : Term1 : f(x,a)

Term2 : f(b,y)

Substitute x with b

Substitute y with a

Final x→b

y→a.

Code: class Var: def
__init__(self, name):
self.name = name

def __repr__(self):
return self.name

def __eq__(self, other):
return isinstance(other, Var) and self.name == other.name

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

```
class Const:     def
__init__(self, name):
        self.name = name

    def __repr__(self):
return self.name

    def __eq__(self, other):
        return isinstance(other, Const) and self.name == other.name

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

class Func:     def __init__(self,
name, args):         self.name =
name
        self.args = args  # list of terms

    def __repr__(self):
        return f"{self.name}({', '.join(map(str, self.args))})"

    def __eq__(self, other):
        return (isinstance(other, Func) and self.name == other.name and
                len(self.args) == len(other.args) and
                all(x == y for x, y in zip(self.args, other.args)))

    def __hash__(self):
        return hash((self.name, tuple(self.args)))

def is_variable(x):
    return isinstance(x, Var)

def is_compound(x):
    return isinstance(x, Func)

def occurs_check(var, term, subst):     if
var == term:         return True     elif
is_variable(term) and term in subst:
        return occurs_check(var, subst[term], subst)
elif is_compound(term):
        return any(occurs_check(var, arg, subst) for arg in term.args)
else:
        return False

def unify(x, y, subst={}, depth=0):
    indent = "  " * depth  # for pretty printing nested calls
if subst is None:         return None     elif x == y:
        # Terms are already identical
```

```
        print(f"{indent}Unify {x} and {y}: terms are identical, no substitution needed")
return subst     elif is_variable(x):
        return unify_var(x, y, subst, depth)
elif is_variable(y):
        return unify_var(y, x, subst, depth)     elif
is_compound(x) and is_compound(y):         if x.name
!= y.name or len(x.args) != len(y.args):
            print(f"{indent}Cannot unify different function symbols or different arity: {x} and {y}")
return None         for x_arg, y_arg in zip(x.args, y.args):
            subst = unify(x_arg, y_arg, subst, depth + 1)
if subst is None:                return None         return
subst     else:
        print(f"{indent}Failed to unify {x} and {y}")
return None

def unify_var(var, x, subst, depth):
    indent = "  " * depth
if var in subst:
        print(f"{indent}Variable {var} already substituted with {subst[var]}, try to unify
{subst[var]} and {x}")         return
unify(subst[var], x, subst, depth + 1)     elif
is_variable(x) and x in subst:
        print(f"{indent}Variable {x} already substituted with {subst[x]}, try to unify {var} and
{subst[x]}")         return unify(var, subst[x],
subst, depth + 1)     elif occurs_check(var, x,
subst):
        print(f"{indent}Occurs check failed: {var} occurs in {x}")
return None     else:
        print(f"{indent}Substitute {var} with {x}")
subst_copy = subst.copy()         subst_copy[var] = x
print(f"{indent}Current substitution: {subst_copy}")
return subst_copy


if __name__ == "__main__":
    # Define terms:
    x = Var('x')     y = Var('y')
f1 = Func('f', [x, Const('a')])
    f2 = Func('f', [Const('b'), y])

    print("Start statement:")
print(f"  Term 1: {f1}")     print(f"
Term 2: {f2}")
    print("\nUnification steps:")

    result = unify(f1, f2, {})

    print("\nEnd state:")
if result is None:
```

```
    print("  Unification failed.")
  else:
    print("   Final substitution:")
for var, val in result.items():
print(f"    {var} -> {val}")
```

Output:

```
Start statement:
  Term 1: f(x, a)
  Term 2: f(b, y)

Unification steps:
  Substitute x with b
  Current substitution: {x: b}
  Substitute y with a
  Current substitution: {x: b, y: a}

End state:
  Final substitution:
    x -> b
    y -> a
```

Program 8:

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

Algorithm:

Week-8 DATE: 15/10/25 PAGE:

Forward Chaining

Pseudocode :

```
function forwardchain (KB, α) :
    inputs = KB, α, query
    local variable : new.
    repeat until new is empty
        new = { }
    for each rule in KB do :
        (p1 ^ p2 ^ ... pn) ← standard-variable (rules)
    for each θ such that substitution (θ, p1^ ... pn^) = substitution (θ, p1^ ... pn^)
        for some p1^ ... pn^ in KB
            q' = subset (θ, q)
```

Q) As per law crime for American to sell weapon to Hostile nations. Country A, enemy of America, has missiles which were sold to it by Robert who is American.

To prove Robert is a criminal

Code:

```
class Predicate:     def
__init__(self, name, args):
        self.name = name         self.args = args  # list
of constants or variables

    def __repr__(self):
        return f"{self.name}({', '.join(map(str, self.args))})"

    def __eq__(self, other):
        return isinstance(other, Predicate) and self.name == other.name and self.args == other.args

    def __hash__(self):
        return hash((self.name, tuple(self.args)))

class Var:     def
__init__(self, name):
        self.name = name

    def __repr__(self):
return self.name

    def __eq__(self, other):
        return isinstance(other, Var) and self.name == other.name

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

class Const:     def
__init__(self, name):
        self.name = name

    def __repr__(self):
return self.name

    def __eq__(self, other):
        return isinstance(other, Const) and self.name == other.name

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

def is_variable(x):
```

```
    return isinstance(x, Var)

def unify(x, y, subst={}):
if subst is None:
return None     elif x ==
y:        return subst
elif is_variable(x): return
unify_var(x, y, subst)
    elif is_variable(y):
        return unify_var(y, x, subst)     elif isinstance(x,
Predicate) and isinstance(y, Predicate):        if x.name !=
y.name or len(x.args) != len(y.args):
            return None         for a, b
in zip(x.args, y.args):
subst = unify(a, b, subst)
if subst is None:                return
None        return subst     else:
        return None

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

def occurs_check(var, x, subst):
if var == x:        return True     elif
is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
elif isinstance(x, Predicate):
        return any(occurs_check(var, arg, subst) for arg in x.args)
else:
        return False

def substitute(predicate, subst):
    new_args = []     for arg
in predicate.args:
        val = arg        while is_variable(val)
and val in subst:
            val = subst[val]
new_args.append(val)
    return Predicate(predicate.name, new_args)
```

```
class Rule:     def __init__(self, premises,
conclusion):        self.premises = premises  #
list of Predicate
        self.conclusion = conclusion  # Predicate

    def __repr__(self):
        return f"{' ∧ '.join(map(str, self.premises))} => {self.conclusion}" def
forward_chain(kb_facts, kb_rules, query):
    inferred          =          set(kb_facts)
print("Initial Facts:")    for f in inferred:
print(f"     {f}")          print("\nStarting
inference steps:\n")

    new_inferred = True

    while new_inferred:
new_inferred = False         for
rule in kb_rules:
            possible_substs = [{}]  # substitutions for premises

            for premise in rule.premises:
                temp_substs = []                for
fact in inferred:                    for subst in
possible_substs:
                        subst_try = unify(premise, fact, subst)
if subst_try is not None:
temp_substs.append(subst_try)
possible_substs = temp_substs

            for subst in possible_substs:
                concluded_fact = substitute(rule.conclusion, subst)                if concluded_fact
not in inferred:                    print(f"Inferred: {concluded_fact} from rule {rule} using
substitution {subst}")                    inferred.add(concluded_fact)                    new_inferred
= True                    if unify(concluded_fact, query) is not None:
                        print(f"\nQuery {query} proved!")
return True

    print(f"\nQuery {query} not proved.")
    return False

if __name__ == "__main__":
    a = Const('a')
b = Const('b')
c = Const('c')
x = Var('x')     y
= Var('y')
    z = Var('z')

    kb_facts = {
```

```
    Predicate('Parent', [a, b]),
    Predicate('Parent', [b, c]),
}

kb_rules = [
    Rule([Predicate('Parent', [x, y])], Predicate('Ancestor', [x, y])),
    Rule([Predicate('Parent', [x, y]), Predicate('Ancestor', [y, z])], Predicate('Ancestor', [x, z])),
]

query = Predicate('Ancestor', [a, c])

print("Running forward chaining...\n")
forward_chain(kb_facts, kb_rules, query)
```

Output:

```
========================================================================== RESTART: C:/Users/STUDENT/Desktop/AI_11
Running forward chaining...

Initial Facts:
  Parent(a, b)
  Parent(b, c)

Starting inference steps:

Inferred: Ancestor(a, b) from rule Parent(x, y) => Ancestor(x, y) using substitution {x: a, y: b}
Inferred: Ancestor(b, c) from rule Parent(x, y) => Ancestor(x, y) using substitution {x: b, y: c}
Inferred: Ancestor(a, c) from rule Parent(x, y) ∧ Ancestor(y, z) => Ancestor(x, z) using substitution {x: a, y: b, z: c}

Query Ancestor(a, c) proved!
```

Program 9:

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

Algorithm:

WEEK-9 DATE: PAGE:

Resolution by First Order Logic

Algorithm:

```
function FOL_RESOLUTION (KB, alpha)
    clauses = convert_to_clauses(KB and not alpha)
    new_clauses = { }
    while True:
        for each (c1, c2) in pairs(clauses):
            resolvent = RESOLVE(c1, c2)
            if resolvent == { }:
                return 'Contradiction found'
                goal is entailed by KB'.

            if resolvent not in new_clauses:
                new_clauses.add(resolvent)

        if new_clauses is subset of clauses:
            return 'No contradiction'.

        clauses = clauses UNION new_clauses.
```

Output: No contradiction, resolved.

Example: Given a) $\forall x \forall y$ Mother$(x,y) \rightarrow$ Parent(x,y)

b) $\forall x \forall y$ (Parent$(x,y) \wedge$ Alive$(x)) \rightarrow$ Older(x,y)

Prove: Older(Leela, Arjun)

Also given: Mother(Leela, Arjun)

Alive(Leela)

Converting to CNF: (a) $\neg$ Mother$(x,y) \vee$ Parent(x,y)

(b) $\neg$ Parent$(x,y) \vee \neg$ Alive$(x) \vee$ Older(x,y)

θ {x/Leela, y/Arjun}

$\therefore \neg$ Mother(Leela, Arjun) $\vee$ Parent(Leela, Arjun)

$\neg$ Parent(Leela, Arjun) $\vee \neg$ Alive(Leela) $\vee$ Older(Leela, Arjun)

Resolution: Negation: $\neg$ Older(Leela, Arjun)

Mother(Leela, Arjun) — $\neg$ Mother(Leela, Arjun) $\vee$ Parent(Leela, Arjun)

Parent(Leela, Arjun)

$\neg$ Parent(Leela, Arjun) $\vee \neg$ Alive(Leela) $\vee$ Older(Leela, Arjun)

$\neg$ Alive(Leela) $\vee$ Older(Leela, Arjun) — Alive(Leela)

$\neg$ Older(Leela, Arjun) — Older(Leela, Arjun)

{ }.

$\therefore$ No contradiction.

goal entailed by KB.

Code:

```
from copy import deepcopy

# ----------------------------
# Basic utilities
# ----------------------------

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

def substitute(subst, expr):
    """Substitute variables in expr according to subst mapping."""
if isinstance(expr, str):
```

```
        return subst.get(expr, expr)
    return (expr[0],) + tuple(substitute(subst, a) for a in expr[1:])

def unify(x, y, subst=None):
    """Unify two literals, returning substitution if possible."""
if subst is None:        subst = {}     if x == y:
        return subst
if is_variable(x):
        return unify_var(x, y, subst)
if is_variable(y):
        return unify_var(y, x, subst)     if isinstance(x, tuple) and isinstance(y, tuple) and
x[0] == y[0] and len(x) == len(y):
        for a, b in zip(x[1:], y[1:]):
            subst = unify(substitute(subst, a), substitute(subst, b), subst)
if subst is None:                return None        return subst
    return None

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

def occurs_check(var, x, subst):
"""Prevent circular substitutions."""
if var == x:        return True     elif
isinstance(x, tuple):
        return any(occurs_check(var, arg, subst) for arg in x[1:])
elif is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
return False

def negate(lit):
    return ('~' + lit[0][1:],) + lit[1:] if lit[0].startswith('~') else ('~' + lit[0],) + lit[1:]

# ----------------------------
# Resolution
# ----------------------------

def resolve(ci, cj):
    """Return resolvents between two clauses."""
resolvents = []
    for li in ci:        for lj in cj:            if li[0].startswith('~') != lj[0].startswith('~') and
li[0].lstrip('~') == lj[0].lstrip('~'):
```

```
            subst = unify(li, negate(lj))
if subst is not None:
                new_clause = set(substitute(subst, l) for l in ci.union(cj) if l != li and l != lj)
resolvents.append(frozenset(new_clause))     return resolvents

def fol_resolution(kb, query):
clauses = deepcopy(kb)
clauses.append({negate(query)})
   print("Initial   clauses:")
for c in clauses:       print("
",                              c)
print("\nResolving...\n")

   new = set()
while True:
       pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1, len(clauses))]
for (ci, cj) in pairs:
          for resolvent in resolve(ci, cj):
if not resolvent:
                print("Derived empty clause -> contradiction found
☑")                return True                new.add(resolvent)        if
new.issubset(set(map(frozenset, clauses))):            print("No new
clauses can be added ✖")            return False         for c in new:
if c not in clauses:
            clauses.append(set(c))
            print("New clause:", c)

# ----------------------------
# Example KB (CNF)
# ----------------------------

KB = [
    {('~Food', 'x'), ('Likes', 'John', 'x')},
    {('Food', 'Apple')},
    {('Food', 'Vegetable')},
    {('~Eats', 'x', 'y'), ('Killed', 'x'), ('Food', 'y')},
    {('Eats', 'Anil', 'Peanut')},
    {('Alive', 'Anil')},
    {('~Eats', 'Anil', 'y'), ('Eats', 'Harry', 'y')},
    {('~Alive', 'x'), ('~Killed', 'x')},
    {('Killed', 'x'), ('Alive', 'x')}
]

query = ('Likes', 'John', 'Peanut')

# ----------------------------
# Run Resolution
# ----------------------------
```

```
proved = fol_resolution(KB, query)

if proved:
    print("\n☑ Proven: John likes peanuts.")
else:
    print("\n❌ Could not prove John likes peanuts.")
```

Output:

```
Initial clauses:
   {('Likes', 'John', 'x'), ('~Food', 'x')}
   {('Food', 'Apple')}
   {('Food', 'Vegetable')}
   {('Food', 'y'), ('~Eats', 'x', 'y'), ('Killed', 'x')}
   {('Eats', 'Anil', 'Peanut')}
   {('Alive', 'Anil')}
   {('~Eats', 'Anil', 'y'), ('Eats', 'Harry', 'y')}
   {('~Alive', 'x'), ('~Killed', 'x')}
   {('Alive', 'x'), ('Killed', 'x')}
   {('~Likes', 'John', 'Peanut')}

Resolving...

New clause: frozenset({('Killed', 'Harry'), ('Food', 'y'), ('~Eats', 'Anil', 'y')})
New clause: frozenset({('Killed', 'Anil'), ('Food', 'Peanut')})
New clause: frozenset({('Killed', 'x'), ('~Killed', 'x')})
New clause: frozenset({('Alive', 'x'), ('~Alive', 'x')})
New clause: frozenset({('Likes', 'John', 'Apple')})
New clause: frozenset({('Likes', 'John', 'Vegetable')})
New clause: frozenset({('~Eats', 'y', 'y'), ('Killed', 'y'), ('Likes', 'John', 'y')})
New clause: frozenset({('~Killed', 'x')})
New clause: frozenset({('Likes', 'John', 'Peanut'), ('Killed', 'Anil')})
New clause: frozenset({('Killed', 'Peanut'), ('~Eats', 'Peanut', 'Peanut')})
New clause: frozenset({('~Eats', 'y', 'y'), ('~Alive', 'y'), ('Likes', 'John', 'y')})
New clause: frozenset({('~Alive', 'Anil'), ('Food', 'Peanut')})
New clause: frozenset({('~Eats', 'Anil', 'y'), ('Killed', 'Harry'), ('Likes', 'John', 'y')})
New clause: frozenset({('~Alive', 'x')})
New clause: frozenset({('~Alive', 'Harry'), ('~Eats', 'Anil', 'y'), ('Food', 'y')})
New clause: frozenset({('Likes', 'John', 'Peanut'), ('~Alive', 'Anil')})
New clause: frozenset({('~Eats', 'Anil', 'Peanut'), ('Killed', 'Harry')})
New clause: frozenset({('Likes', 'John', 'Peanut')})
New clause: frozenset({('Killed', 'Anil')})
New clause: frozenset({('~Eats', 'Peanut', 'Peanut')})
New clause: frozenset({('~Eats', 'Anil', 'y'), ('Likes', 'John', 'y')})
New clause: frozenset({('~Alive', 'Peanut'), ('~Eats', 'Peanut', 'Peanut')})
New clause: frozenset({('~Eats', 'Anil', 'y'), ('~Alive', 'Harry'), ('Likes', 'John', 'y')})
Derived empty clause -> contradiction found ☑

☑ Proven: John likes peanuts.
```

Program 10:
Implement Alpha-Beta Pruning.

Algorithm:

WEEK-10

ALPHA - BETA PRUNING.

```
function alpha-beta-search(state) returns an action
    v ← max-value(state, -∞, ∞)
    return action in ACTIONS(state) with value v.

function max-value(state, α, β) returns α utility value
    if terminal-test(state) then return utility(state)
    v ← -∞
    for each α in action(state) do
        v ← max(v, min-value(result(s,a), α, β))
        if v ≥ β return v
        α ← max(α, v)
    return v.

function min-value(state, α, β) returns α utility value
    if terminal-test(state) then return utility(state)
    v ← +∞
    for each α 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: Max Value : 10

Code:
```
import math

# ==================================================================
# Define a simple binary game tree
# ==================================================================
```

```
class Node:     def __init__(self, value=None,
children=None):         self.value = value          #
Leaf value (for terminal nodes)         self.children
= children or []  # List of child nodes


# ============================================================
# Alpha–Beta Pruning Algorithm
# ============================================================

def alpha_beta(node, depth, alpha, beta, maximizing_player):
    """
    Alpha–Beta pruning on an explicit tree structure.
    """
    # Base case: leaf node     if depth
== 0 or not node.children:
        return node.value

    if maximizing_player:
max_eval = -math.inf         for
child in node.children:
            eval = alpha_beta(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
alpha = max(alpha, eval)               if beta
<= alpha:                        break   # prune
return max_eval     else:
        min_eval = math.inf
for child in node.children:
            eval = alpha_beta(child, depth - 1, alpha, beta, True)
min_eval = min(min_eval, eval)                beta = min(beta,
eval)            if beta <= alpha:                break  # prune
        return min_eval


# ============================================================
# Trace version (to see pruning steps)
# ============================================================

def alpha_beta_trace(node, depth, alpha, beta, maximizing_player, indent=""):
if depth == 0 or not node.children:
        print(f"{indent}Leaf → value = {node.value}")
        return node.value

    if maximizing_player:         print(f"{indent}MAX
node, α={alpha}, β={beta}")
        value = -math.inf         for i, child in
enumerate(node.children):
```

```
        print(f"{indent}→ Explore MIN child {i}")            eval =
alpha_beta_trace(child, depth - 1, alpha, beta, False, indent + "  ")            value
= max(value, eval)
        alpha = max(alpha, value)            print(f"{indent}MAX updated:
value={value}, α={alpha}, β={beta}")            if beta <= alpha:
            print(f"{indent}PRUNE remaining children (β ≤ α)\n")
break        return value    else:
      print(f"{indent}MIN node, α={alpha}, β={beta}")
value = math.inf        for i, child in
enumerate(node.children):
        print(f"{indent}→ Explore MAX child {i}")            eval =
alpha_beta_trace(child, depth - 1, alpha, beta, True, indent + "  ")            value
= min(value, eval)            beta = min(beta, value)            print(f"{indent}MIN
updated: value={value}, α={alpha}, β={beta}")            if beta <= alpha:
            print(f"{indent}PRUNE remaining children (β ≤ α)\n")
break
      return value


# ============================================================
# Example Tree (Depth 3, 8 leaf nodes)
# ============================================================

# Leaves
leaf_nodes = [Node(3), Node(5), Node(6), Node(9), Node(1), Node(2), Node(0), Node(-1)]

# Level 2 (MIN nodes)
level2 = [
   Node(children=[leaf_nodes[0], leaf_nodes[1]]),
   Node(children=[leaf_nodes[2], leaf_nodes[3]]),
   Node(children=[leaf_nodes[4], leaf_nodes[5]]),
   Node(children=[leaf_nodes[6], leaf_nodes[7]])
]

# Level 1 (MAX nodes)
level1 = [
   Node(children=[level2[0], level2[1]]),
   Node(children=[level2[2], level2[3]])
]

# Root (MIN node)
root = Node(children=[level1[0], level1[1]])


# ============================================================
# Run the algorithm
# ============================================================

if __name__ == "__main__":
```

```
    print("========== Alpha–Beta Pruning (No Trace) ==========")
    optimal_value = alpha_beta(root, 3, -math.inf, math.inf, True)
print(f"\n☑ Optimal value found: {optimal_value}")

    print("\n========== Alpha–Beta Pruning (Trace) ==========\n")
traced_value = alpha_beta_trace(root, 3, -math.inf, math.inf, True)
print(f"\n☑ Final optimal value (trace) = {traced_value}")
```

Output:

```
========== Alpha-Beta Pruning (No Trace) ==========

☑ Optimal value found: 5

========== Alpha-Beta Pruning (Trace) ==========

MAX node, α=-inf, β=inf
→ Explore MIN child 0
  MIN node, α=-inf, β=inf
  → Explore MAX child 0
    MAX node, α=-inf, β=inf
    → Explore MIN child 0
      Leaf → value = 3
    MAX updated: value=3, α=3, β=inf
    → Explore MIN child 1
      Leaf → value = 5
    MAX updated: value=5, α=5, β=inf
  MIN updated: value=5, α=-inf, β=5
  → Explore MAX child 1
    MAX node, α=-inf, β=5
    → Explore MIN child 0
      Leaf → value = 6
    MAX updated: value=6, α=6, β=5
    PRUNE remaining children (β ≤ α)

  MIN updated: value=5, α=-inf, β=5
MAX updated: value=5, α=5, β=inf
→ Explore MIN child 1
  MIN node, α=5, β=inf
  → Explore MAX child 0
    MAX node, α=5, β=inf
    → Explore MIN child 0
      Leaf → value = 1
    MAX updated: value=1, α=5, β=inf
    → Explore MIN child 1
      Leaf → value = 2
    MAX updated: value=2, α=5, β=inf
  MIN updated: value=2, α=5, β=2
  PRUNE remaining children (β ≤ α)

MAX updated: value=5, α=5, β=inf

☑ Final optimal value (trace) = 5
```