

RAJALAKSHMI ENGINEERING COLLEGE



**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE
LEARNING**

AI23231-PRINCIPLES OF ARTIFICIAL INTELLIGENCE LAB

[Regulation 2023]

I Year – II Semester

LABORATORY MANUAL



RAJALAKSHMI ENGINEERING COLLEGE

VISION & MISSION OF RAJALAKSHMI ENGINEERING COLLEGE

Vision

To be an institution of excellence in Engineering, Technology and Management Education & Research. To provide competent and ethical professionals with a concern for society.

Mission

To impart quality technical education imbued with proficiency and humane values. To provide right ambience and opportunities for the students to develop into creative, talented and globally competent professionals. To promote research and development in technology and management for the benefit of the society.



RAJALAKSHMI ENGINEERING COLLEGE
B. E. ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

VISION:

- To promote highly Ethical and Innovative Computer Professionals through excellence in teaching, training and research.

MISSION:

- To produce globally competent professionals, motivated to learn the emerging technologies and to be innovative in solving real world problems.
- To promote research activities amongst the students and the members of faculty that could benefit the society.
- To impart moral and ethical values in their profession.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO 1: Graduates will demonstrate their technical skills and competency in various applications through the use of Artificial Intelligence and Data Science.

PEO 2: To produce motivated graduates with capability to apply acquired knowledge and skills in data analytics and visualization to develop viable systems.

PEO 3: Graduates will establish their knowledge by adopting Artificial Intelligence and Data Science technologies to solve the real world problems

PEO 4: To produce graduates with potential to participate in life-long learning through professional developments for societal needs with ethical values.

PROGRAMME OUTCOMES (POs)

PO1: Engineering knowledge: Apply the knowledge of Mathematics, Science, Engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO 4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO 5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO 6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO 8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

A graduate of the Artificial Intelligence and Data Science Learning Program will demonstrate.

PSO 1: Foundation Skills: Apply computing theory, languages and algorithms, as well as mathematical and statistical models, and the principles of optimization to appropriately formulate and use data analysis.

PSO 2: Problem-Solving Skills: The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product for business automation.

PSO 3: Successful Progression: Ability to critique the role of information and analytics for an innovative career, research activities and consultancy.

Syllabus

Subject Code	Subject Name(Lab oriented Theory Courses)	Category	L	T	P	C
AI23231	PRINCIPLES OF ARTIFICIAL INTELLIGENCE	PC	3	0	2	4
Objectives:						
●	To Understand the various characteristics of a problem solving agent					
●	To Learn about the different strategies involved in problem solving.					
●	To Learn about solving problems with various constraints.					
●	To Learn about various knowledge representation					
●	To Understand the different models of reasoning and decision making					
UNIT-I	INTRODUCTION TO ARTIFICIAL INTELLIGENCE AND PROBLEM-SOLVING AGENT					9
AI-Introduction. Intelligent Agents, Agents & environment, nature of environment, structure of agents, goal-based agents, utility-based agents, learning agents. Defining the problem as state space search, production system, problem characteristics, issues in the design of search programs.						
UNIT-II	SEARCH TECHNIQUES					9
Problem solving agents, searching for solutions; uniform search strategies: breadth first search, depth first search, depth limited search, bidirectional search. Heuristic search strategies Greedy best-first search, A* search, AO* search, memory bounded heuristic search: local search algorithms & optimization problems: Hill climbing search, simulated annealing search, local beam search.						
UNIT-III	CONSTRAINT SATISFACTION PROBLEMS AND GAME THEORY					9
Local search for constraint satisfaction problems. Adversarial search, Games, optimal decisions & strategies in games, the min max search procedure, alpha-beta pruning.						
UNIT-IV	KNOWLEDGE REPRESENTATION					9
AI for knowledge representation, rule-based knowledge representation, procedural and declarative knowledge, Logic programming, Forward and backward reasoning.						
UNIT-V	REASONING & DECISION MAKING					9

Statistical Reasoning: Probability and Bays' Theorem, Certainty Factors and Rule-Base Systems, Bayesian Networks, Dempster-Shafer Theory, Fuzzy Logic. Decision networks, Markov Decision Process. Expert System

Contact Hours	:	45
------------------	---	----

List of Experiments

- | | |
|---|--|
| 1 | <p>Programs on Problem Solving</p> <ul style="list-style-type: none"> a. Write a program to solve 8 Queens problem b. Solve any problem using depth first search c. Implement MINIMAX algorithm d. Implement A* algorithm |
| 2 | <p>Programs on Decision Making and Knowledge Representation</p> <ul style="list-style-type: none"> a. Introduction to PROLOG <ul style="list-style-type: none"> i) Find minimum maximum of two numbers ii) Here are some simple clauses. <pre>likes(mary,food). likes(mary,wine). likes(john,wine). likes(john,mary).</pre> <p>The following queries yield the specified answers.</p> <pre> ?- likes(mary,food). yes. ?- likes(john,wine). yes. ?- likes(john,food). no.</pre> <p>How do you add the following facts?</p> |

	1. John likes anything that Mary likes 2. John likes anyone who likes wine 3. John likes anyone who likes themselves	
	b. Implementation of Unification and Resolution Algorithm	
	c. Implementation of Backward Chaining	
	d. Implementation of Forward Chaining	
3	Programs on Planning and Learning	
	a. Implementation of Blocks World program	
	b. Implementing a fuzzy inference system	
Requirements		
Hardware	Intel i3, CPU @ 1.20GHz 1.19 GHz, 4 GB RAM, 32 Bit Operating System	
Software	C or Python Knowledge representation experiments can be performed using a PROLOG TOOL	
Operating System	Windows	

AI23231 – ARTIFICIAL INTELLIGENCE LAB PLAN

Exercise No.	Exercise Name	Required Hours
1	Write a program to solve 8 Queens problem	2
2	Solve any problem using depth first search	2
3	Implementation of MINIMAX algorithm	2
4	Implementation of A* algorithm	2
5	Find minimum maximum of two numbers	1
6	Implementation of Simple Clause likes(mary,food). likes(mary,wine). likes(john,wine). likes(john,mary). The following queries yield the specified answers. ?- likes(mary,food). yes. ?- likes(john,wine). yes. ?- likes(john,food). no. How do you add the following facts? 1. John likes anything that Mary likes 2. John likes anyone who likes wine 3. John likes anyone who likes themselves	1
7	Implementation of Unification and Resolution Algorithm	2
8	Implementation of Backward Chaining	2
9	Implementation of Forward Chaining	2
10	Implementation of Blocks World program	2
11	Implementing a fuzzy inference system	2

Course Outcomes (COs)**Course Name: Artificial Intelligence Lab****Course Code: AI23231**

Outcome 1	Basic knowledge representation, problem solving, and learning methods of artificial intelligence.
Outcome 2	Provide the apt agent strategy to solve a given problem
Outcome 3	Represent a problem using first order and predicate logic
Outcome 4	Design applications like expert systems and chat-bot.
Outcome 5	Suggest the different models of reasoning and decision making for any given problem

CO-PO –PSO matrices of course

PO/PSO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
CS23231.1	3	3	1	-	2	1	1	1	1	-	2	1	2	1	1
CS23231.2	2	2	1	-	2	1	2	-	-	-	2	2	1	1	1
CS23231.3	3	3	1	-	3	-	1	-	-	-	3	1	2	3	2
CS23231.4	2	2	2	2	3	-	1	2	-	-	2	2	2	2	2
CS23231.5	2	3	-	-	2	1	1	1	-	-	2	2	2	2	2
Average	2.4	2.4	1.8	2.0	2.4	0.6	1.2	0.8	0.2	-	2.0	1.8	1.8	1.8	1.6

Note: Enter correlation levels 1,2 or 3 as defined below:

1: Slight(Low) 2: Moderate(Medium) 3: Substantial(High) If there is no correlation, put “-”

CO-PO-PSO JUSTIFICATION

CO-PO/PSO	Level	Justification
CO1-PO1	3	Strongly correlated as CO1 builds foundational knowledge in AI and its methodologies.
CO1-PO2	3	Strongly supports problem analysis, as AI techniques are used to solve problems efficiently.
CO1-PO3	1	Weak correlation since design and development of AI models is not the primary focus.
CO1-PO4	-	No correlation
CO1-PO5	2	Moderate correlation as some modern AI tools are introduced but not deeply covered.
CO1-PO6	1	Weak correlation as sustainability aspects are not the primary focus.
CO1-PO7	1	Weak correlation because ethical considerations are minor.
CO1-PO8	1	Weak correlation due to limited societal impact.
CO1-PO9	1	Weak correlation due to limited focus on team work
CO1-P10	-	No correlation
CO1-P11	2	Moderate correlation as project management skills are involved.
CO1-P12	1	Weak correlation with lifelong learning since basic AI concepts form a foundation for further study.
CO1-PSO1	2	Moderate correlation with domain-specific AI knowledge and applications.
CO1-PSO2	1	Weak correlation due to limited practical application.
CO1-PSO3	1	Weak correlation since applied AI techniques are only slightly covered.
CO2-PO1	2	Moderate correlation, as AI agents require fundamental problem-solving techniques.
CO2-PO2	2	Moderate correlation with analytical skills, as strategies are predefined.
CO2-PO3	1	Moderate correlation with design and development.
CO2-PO4	-	No correlation with investigation aspects.
CO2-PO5	2	Moderate correlation since engineering tools are used.
CO2-PO6	1	Weak correlation due to minimal environmental considerations.
CO2-PO7	2	Moderate correlation as ethical factors play a role.

CO2-PO8	-	No correlation
CO2-PO9	-	No correlation
CO2-P10	-	No correlation
CO2-P11	2	Moderate correlation with modern tools.
CO2-P12	2	Moderate correlation with lifelong learning since AI strategies evolve with time.
CO2-PSO1	1	Weak correlation as intelligent agent strategies align with industry AI practices.
CO2-PSO2	1	Weak correlation, as domain-specific applications are not deeply explored.
CO2-PSO3	1	Weak Correlation as students can limit building effective AI-based solutions.
CO3-PO1	3	Strong correlation, as logical representation is fundamental in AI.
CO3-PO2	3	Strong correlation with problem analysis using formal logic methods.
CO3-PO3	1	Weak correlation, as problem representation does not involve design aspects.
CO3-PO4	-	No correlation
CO3-PO5	3	Strong correlation, as formal logic is applied using AI-based tools.
CO3-PO6	-	No correlation
CO3-PO7	1	Weak correlation with ethics.
CO3-PO8	-	No correlation
CO3-PO9	-	No correlation
CO3-P10	-	No correlation
CO3-P11	3	Strong correlation with modern tools and applications
CO3-PO12	1	Weak correlation with lifelong learning since logic representation is foundational.
CO3-PSO1	2	Strong correlation with domain-specific AI logic and problem representation techniques.
CO3-PSO2	3	Moderate correlation, as logic plays a role in applied AI applications.
CO3-PSO3	2	Moderate correlation with students gain the skills needed to design and develop AI-based solutions for real-world challenges.
	2	Moderate correlation, as AI knowledge is applied to real-

CO4-PO1		world applications.
CO4-PO2	2	Moderate correlation with problem analysis and designing intelligent systems.
CO4-PO3	2	Moderate correlation, as AI systems require structured design methodologies.
CO4-PO4	2	Investigative methods are employed to test and analyze graph algorithms for correctness and efficiency.
CO4-PO5	3	Strong correlation, as AI development involves the use of modern tools and frameworks.
CO4-PO6	-	No correlation
CO4-PO7	1	Moderate correlation, as ethical concerns in AI applications are addressed.
CO4-PO8	2	By understanding knowledge representation and its ethical implications, students develop AI solutions that align with professional ethics and societal responsibilities
CO4-PO9	-	No correlation
CO4-PO10	-	No correlation
CO4-PO11	2	Moderate correlation, as project management is needed for AI system development.
CO4-P12	2	Students must continuously update their knowledge, learn new technologies, and adapt to advancements in AI, such as natural language processing, deep learning, and ethical AI principles.
CO4-PSO1	2	Moderate correlation with technical skills required for AI-based applications.
CO4-PSO2	2	Moderate correlation with domain-specific AI applications.
CO4-PSO3	2	Moderate correlation with industry-focused AI system development.
CO5-PO1	2	Strong correlation, as decision-making models are fundamental AI techniques.
CO5-PO2	3	Strong correlation with problem-solving and analysis in AI.
CO5-PO3	-	No correlation
CO5-PO4	-	No correlation
CO5-PO5	2	Moderate correlation, as AI reasoning models are applied using modern tools.
CO5-PO6	1	By learning how AI systems reason and make decisions, students can analyze the societal and ethical impacts of AI solutions, ensuring they are fair, safe, and beneficial to society.
	1	Weak correlation, as ethical concerns in decision-making

CO5-PO7		are indirectly involved.
CO5-PO8	1	students can design AI systems that follow ethical guidelines, avoid bias, ensure transparency, and uphold professional responsibilities
CO5-PO9	-	No correlation
CO5-PO10	-	No correlation
CO5-PO11	2	students can optimize resources, assess risks, and make informed financial and managerial decisions, ensuring the successful execution of AI projects
CO5-P12	2	Moderate correlation with lifelong learning since AI reasoning models evolve over time.
CO5-PSO1	2	Moderate correlation with domain-specific AI decision-making techniques.
CO5-PSO2	2	Moderate correlation, as decision-making models are used in various AI applications.
CO5-PSO3	2	Moderate correlation with industry-focused AI solutions.

Faculty-in-charge

Mentor

HoD

Ex No: 1

IMPLEMENT EIGHT QUEENS PROBLEM

Aim:

To develop a Python program that solves the **8-Queens problem** using the **Backtracking algorithm**. The program should ensure that **no two queens attack each other** and display valid chessboard configurations.

Case Scenario:

A chessboard consists of **8×8 squares**, and your task is to place **8 queens** on the board such that **no two queens attack each other**. Queens can attack in **horizontal, vertical, and diagonal** directions.

Task Requirements:

1. **Problem Representation:**
 - Represent the **8-queens problem** as a **constraint satisfaction problem (CSP)** or a **search problem**.
2. **Algorithm Implementation:**
 - Implement a solution using either **Backtracking** or **Genetic Algorithm**.
3. **Output Requirements:**
 - Display a valid **8×8 chessboard** with queens (Q) placed correctly.
 - Show multiple valid solutions if possible.
4. **Performance Analysis:**
 - Compare execution time for different **board sizes (e.g., 4×4, 8×8, 10×10)**.

Procedure:

1. **Start**
2. **Initialize an N×N chessboard with all empty positions (.).**
3. **Define a function `is_safe(board, row, col, N)`:**
 - Check if placing a queen at `(row, col)` violates any constraints.
4. **Define a recursive function `solve_n_queens(board, row, N)`:**
 - If `row == N`, print the board (solution found).
 - Try placing a queen in each column (0 to N-1).
 - If `is_safe() == True`, place the queen and recurse for the next row.
 - If placing a queen leads to failure, backtrack (remove the queen).
5. **Call `solve_n_queens()` for the first row (`row = 0`).**
6. **If a solution is found, print the board; else, print "No solution exists."**
7. **End**

Program

```
import copy
```

```

N = 8 # Size of the chessboard (8x8)

# Function to print the solution
def printSolution(board):
    for row in board:
        for i in range(N):
            print("Q" if row[i] else ".", end=" ")
        print()
    print() # Add a newline for readability

# Function to check if a queen can be placed on board[row][col]
def isSafe(board, row, col):
    # Check the column
    for i in range(row):
        if board[i][col]:
            return False

    # Check the upper left diagonal
    for i, j in zip(range(row - 1, -1, -1), range(col - 1, -1, -1)):
        if board[i][j]:
            return False

    # Check the upper right diagonal
    for i, j in zip(range(row - 1, -1, -1), range(col + 1, N)):
        if board[i][j]:
            return False
    return True

# Function to solve the 8 Queens problem using backtracking
def solve(board, row, solutions):
    if row == N:

```



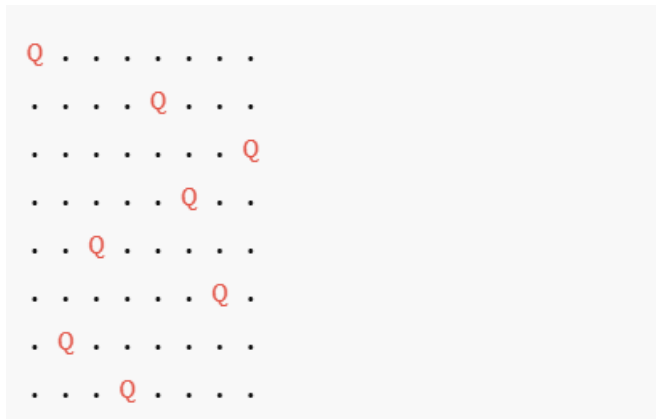
```

        solutions.append(copy.deepcopy(board)) # Deep copy of the board
        printSolution(board)
        return

    for col in range(N):
        if isSafe(board, row, col):
            board[row][col] = 1 # Place queen
            solve(board, row + 1, solutions) # Recur to place next queen
            board[row][col] = 0 # Backtrack (remove queen)
# Main function to initialize the board and start solving the problem
def eightQueens():
    board = [[0 for _ in range(N)] for _ in range(N)]
    solutions = [] # Store all solutions
    solve(board, 0, solutions)
    print(f"Total solutions found: {len(solutions)}")
# Calling the function
eightQueens()

```

Output:



```

Q . . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . Q .
. Q . . . . . .
. . . Q . . . .

```

Ex No: 1b

IMPLEMENTATION OF DEPTH FIRST SEARCH

Aim:

To implement depth first search

Case Scenario:

A **robotic delivery system** is implemented in a smart warehouse. The warehouse is modeled as a **graph**, where each **node represents a storage unit** and each **edge represents a possible path**. The robot needs to pick up a package from a starting point and deliver it to the correct storage location.

The **robot's movement strategy** is to explore the storage units **by going as deep as possible before backtracking** if needed. The **warehouse is not fully mapped**, so the robot uses a **Depth First Search (DFS) algorithm** to explore the paths.

Procedure:

Step 1: Input the Graph

- Represent the warehouse as a **graph (Adjacency List)**.
- Define the **start node** and **goal node**.

Step 2: Initialize DFS

- Use a **set (visited)** to track visited nodes.
- Use a **list (path)** to store the current traversal path.

Step 3: Recursive DFS Function

1. **Mark the current node as visited.**
2. **Add the current node to the path.**
3. **Check if the current node is the goal:**
 - If **yes**, return the path.
 - If **no**, proceed with the next steps.
4. **Explore all neighboring nodes:**
 - If a neighbor is **not visited**, recursively call DFS on it.
 - If a valid path is found, return it.
5. **If no path is found, return None.**

Step 4: Call the DFS Function

- Call DFS with the given **start** and **goal nodes**.
- Print the path found (if any).

Program:

Depth First Search (DFS) implementation for a warehouse graph

```

# Sample warehouse graph as an adjacency list

warehouse_graph = {

    'A': ['B', 'C'],

    'B': ['D', 'E'],

    'C': ['F'],

    'D': [],

    'E': ['F'],

    'F': []

}

# Function to perform DFS

def dfs(graph, start, goal, visited=None, path=None):

    if visited is None:

        visited = set()

    if path is None:

        path = []

    # Mark current node as visited and add to path

    visited.add(start)

    path.append(start)

    # If goal is found, return the path

    if start == goal:

        return path

    # Explore neighbors

    for neighbor in graph[start]:

```

```
    if neighbor not in visited:

        result = dfs(graph, neighbor, goal, visited, path[:]) # Use path[:] to copy path

    if result: # Stop if a path is found

        return result

    return None # No path found

# Example usage

start_node = 'A'

goal_node = 'F'

path_found = dfs(warehouse_graph, start_node, goal_node)

print(f'DFS Path from {start_node} to {goal_node}: {path_found}')
```

Output:

```
DFS Path from A to F: ['A', 'B', 'E', 'F']
```

Or

```
DFS Path from A to F: ['A', 'C', 'F']
```

Ex No: 1c

IMPLEMENTATION OF MINIMAX algorithm

Aim:

To implement MINIMAX algorithm.

Scenario: AI vs. Human Player – Winning Move Situation

Context: **The AI is playing as Player X and the human is playing as Player O. It's AI's turn, and there is a possible winning move.**

Given Board State (Before AI's Move):

```
X O X
O X .
. O X
```

Expected AI Move (Best Move using Minimax):

```
X O X
O X X
. O X
```

Procedure:

1. Define constants: `PLAYER_X = 1, PLAYER_O = -1, EMPTY = 0`.
2. Create `evaluate(board)` to check for a winner by scanning rows, columns, and diagonals. Return 1 if AI wins, -1 if human wins, or 0 if no winner.
3. Create `isMovesLeft(board)` to check for empty spaces; return True if moves are available, otherwise False.
4. Implement `minimax(board, isMax)`:
 - If `evaluate(board)` returns a winner, return the corresponding score.
 - If no moves are left, return 0.
 - If `isMax` is True (AI's turn), initialize `best = -∞`, loop through empty cells, place X, call `minimax(board, False)`, undo move, update `best` with maximum value, and return `best`.
 - If `isMax` is False (Human's turn), initialize `best = +∞`, loop through empty cells, place O, call `minimax(board, True)`, undo move, update `best` with minimum value, and return `best`.
5. Implement `findBestMove(board)`:
 - Initialize `bestVal = -∞` and `bestMove = (-1, -1)`.
 - Loop through empty cells, place X, call `minimax(board, False)`, undo move, update `bestMove` if a better move is found.
 - Return `bestMove`.
6. Implement `printBoard(board)` to display board state using "X", "O", and "." for empty spaces.

7. Initialize a sample board, print its state, call `findBestMove(board)`, update the board with AI's move, and print the final state.

Program:

```
# Constants for players
```

```
PLAYER_X = 1
```

```
PLAYER_O = -1
```

```
EMPTY = 0
```

```
# Evaluate the board
```

```
def evaluate(board):
```

```
    for row in range(3):
```

```
        if board[row][0] == board[row][1] == board[row][2] != EMPTY:
```

```
            return board[row][0]
```

```
    for col in range(3):
```

```
        if board[0][col] == board[1][col] == board[2][col] != EMPTY:
```

```
            return board[0][col]
```

```
    if board[0][0] == board[1][1] == board[2][2] != EMPTY:
```

```
        return board[0][0]
```

```
    if board[0][2] == board[1][1] == board[2][0] != EMPTY:
```

```
        return board[0][2]
```

```
    return 0
```

```
# Check if moves are left
```

```
def isMovesLeft(board):
```

```
    for row in range(3):
```

```
        for col in range(3):
```

```
            if board[row][col] == EMPTY:
```

```
                return True
```

```
    return False
```

```
# Minimax function
```

```
def minimax(board, isMax):
```

```
    score = evaluate(board)
```

```
    if score == PLAYER_X: return score
```

```

if score == PLAYER_O: return score
if not isMovesLeft(board): return 0
if isMax:
    best = -float('inf')
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_X
                best = max(best, minimax(board, not isMax))
                board[row][col] = EMPTY
    return best
else:
    best = float('inf')
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_O
                best = min(best, minimax(board, not isMax))
                board[row][col] = EMPTY
    return best
# Find the best move for PLAYER_X
def findBestMove(board):
    bestVal = -float('inf')
    bestMove = (-1, -1)
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_X
                moveVal = minimax(board, False)
                board[row][col] = EMPTY
                if moveVal > bestVal:
                    bestMove = (row, col)

```

```

        bestVal = moveVal

    return bestMove

# Print the board
def printBoard(board):
    for row in board:
        print(" ".join(["X" if x == PLAYER_X else "O" if x == PLAYER_O else "." for x in
row]))

# Example game
board = [
    [PLAYER_X, PLAYER_O, PLAYER_X],
    [PLAYER_O, PLAYER_X, EMPTY],
    [EMPTY, PLAYER_O, PLAYER_X]
]

print("Current Board:")
printBoard(board)

move = findBestMove(board)
print(f"Best Move: {move}")

board[move[0]][move[1]] = PLAYER_X
print("\nBoard after best move:")
printBoard(board)

```

Output:

```

Current Board:
X O X
O X .
. O X

Best Move: (2, 0)

Board after best move:
X O X
O X .
X O X

```

Ex No: 1d

IMPLEMENTATION OF A* SEARCH ALGORITHM

Aim:

To implement A* Search Algorithm.

Case Scenario:

A delivery robot in a warehouse needs to find the shortest path from the entrance to a specific package location. The warehouse is represented as a grid with some obstacles. Implement the A* search algorithm to help the robot navigate efficiently.

Procedure:

1. **Define the Node class** with attributes: position, parent, cost (g), heuristic (h), and total cost ($f = g + h$).

2. **Implement the heuristic function** using the Manhattan distance formula.

3. **Initialize A Search* with:**

- `open_list` (priority queue) containing the start node.
- `closed_set` to store visited nodes.

4. **While `open_list` is not empty:**

- Extract the node with the lowest `f-value`.
- If the goal is reached, trace back the path and return it.
- Add the current node to `closed_set`.
- Generate new valid moves (up, down, left, right) ensuring they are within bounds and not obstacles.
- Calculate new cost `g`, heuristic `h`, and total `f`.
- Add new nodes to `open_list` for further exploration.

5. **Return the optimal path** if found, else return `None` if no path exists.

Program:

```
import heapq

# Define the grid and movements

class Node:

    def __init__(self, position, parent=None, g=0, h=0):

        self.position = position # (row, col)

        self.parent = parent # Parent node
```

```

    self.g = g # Cost from start node
    self.h = h # Heuristic cost to goal
    self.f = g + h # Total cost
    def __lt__(self, other):
        return self.f < other.f # Priority queue comparison
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1]) # Manhattan Distance
def a_star(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    open_list = []
    heapq.heappush(open_list, Node(start, None, 0, heuristic(start, goal)))
    closed_set = set()
    while open_list:
        current_node = heapq.heappop(open_list) # Get node with lowest f-value
        if current_node.position == goal:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1] # Return reversed path

        closed_set.add(current_node.position)
        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]: # Possible moves
            new_pos = (current_node.position[0] + dr, current_node.position[1] + dc)
            if (0 <= new_pos[0] < rows and 0 <= new_pos[1] < cols and
                grid[new_pos[0]][new_pos[1]] == 0 and new_pos not in closed_set):
                new_node = Node(new_pos, current_node, current_node.g + 1, heuristic(new_pos,
goal))
                heapq.heappush(open_list, new_node)
    return None # No path found

# Example grid: 0 = free space, 1 = obstacle

```

```
warehouse_grid = [  
    [0, 0, 0, 0, 1],  
    [1, 1, 0, 1, 0],  
    [0, 0, 0, 0, 0],  
    [0, 1, 1, 1, 0],  
    [0, 0, 0, 0, 0]  
]  
start_position = (0, 0)  
goal_position = (4, 4)  
path = a_star(warehouse_grid, start_position, goal_position)  
print("Optimal Path:", path)
```

Output:

```
Optimal Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)]
```

Ex No: 2

IMPLEMENTATION OF DECISION MAKING AND KNOWLEDGE REPRESENTATION

Aim:

To implement decision making and knowledge representation using prolog tool.

Prolog Code:

% Rule to find the minimum of two numbers

minimum(X, Y, X) :- X <= Y. % If X is less than or equal to Y, X is the minimum.

minimum(X, Y, Y) :- X > Y. % If X is greater than Y, Y is the minimum.

% Rule to find the maximum of two numbers maximum(X, Y, X) :- X >= Y. % If X is greater than or equal to Y, X is the maximum. maximum(X, Y, Y) :- X < Y. % If X is less than Y, Y is the maximum.

Example Queries:

1. To find the minimum of two numbers:

```
?- minimum(5, 10, Min).
```

Output:

Min = 5.

2. To find the maximum of two numbers:

```
?- maximum(5, 10, Max).
```

Output:

Max = 10.

```
?- minimum(8, 3, Min), maximum(8, 3, Max).
```

Output:

Min = 3, Max = 8.

Prolog Code:

% Given facts

likes(mary, food).

likes(mary, wine).

likes(john, wine).

likes(john, mary).

% Rules based on the conditions:

likes(john, X) :- likes(mary, X). % John likes anything that Mary likes

likes(john, Y) :- likes(Y, wine). % John likes anyone who likes wine

likes(john, Y) :- likes(Y, Y). % John likes anyone who likes themselves

% Sample queries:

% Query 1: Does John like food?

% ?- likes(john, food).

% Query 2: Does John like wine?

% ?- likes(john, wine).

% Query 3: Does John like food if Mary likes food?

% ?- likes(john, food).

% Query 4: Who does John like?

% ?- likes(john, Y).

Output:

Query: ?- likes(john, food).

yes

Query: ?- likes(john, wine).

yes

Query: ?- likes(john, food).

yes

Query: ?- likes(john, Y).

Y = mary ;

Y = john ;

Y = wine ;

Query?- likes(john, Y).

Y = mary ;

Y = john ;

Y = wine ;

Ex No: 2b

IMPLEMENTATION OF UNIFICATION AND RESOLUTION

ALGORITHM

Aim:

To implement unification and resolution algorithm using python.

Scenario:

In an AI-based expert system for **automated reasoning**, the system needs to resolve queries by **unifying logical predicates** and applying **resolution inference**. For example, given the knowledge base:

- **Rule 1:** If John is a **human**, then John is a **mortal** $\rightarrow \text{Human}(\text{John}) \rightarrow \text{Mortal}(\text{John})$
- **Fact 1:** $\text{Human}(\text{John})$
- **Query:** Is John mortal?

Procedure:

1. Define the unification function (**unify**):

- If both terms are identical, return the current substitution (θ).
- If one term is a **variable**, unify it with the other term.
- If both terms are **compound expressions**, unify their corresponding parts recursively.
- Otherwise, return **None** (unification fails).

2. Define the variable unification function (**unify_var**):

- If the variable already exists in the substitution set, apply unification recursively.
- Otherwise, assign the variable to the given term.

3. Define the resolution function (**resolution**):

- Iterate through the **knowledge base** (KB).
- Try to **unify** the given query with KB clauses.
- If unification succeeds, remove matched parts from KB and **recurse with the remaining** parts.
- If the knowledge base is empty after resolution, **the query is proven**.
- Otherwise, return **False** (query not proven).

4. Provide a knowledge base with facts and implications.

5. Define a query to resolve (e.g., **Mortal(John)**).

6. **Run the resolution function to check if the query can be proven.**
7. **Print whether the query is resolved.**

Program:

```
import re
# Function to check if two predicates can be unified
def unify(x, y, theta={}):
    if theta is None:
        return None
    elif x == y:
        return theta
    elif isinstance(x, str) and x.islower(): # x is a variable
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.islower(): # y is a variable
        return unify_var(y, x, theta)
    elif isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        return unify(x[1:], y[1:], unify(x[0], y[0], theta))
    else:
        return None
# Function to unify a variable with a term
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
# Function to apply resolution rule
def resolution(kb, query):
    for clause in kb:
        theta = unify(clause[0], query, {})
```



```

    if theta is not None:
        new_kb = clause[1:]
        if not new_kb: # If empty, means query is resolved
            return True
        else:
            return resolution(kb, new_kb[0])
    return False
# Knowledge base (Implications)
knowledge_base = [
    ["Human", "John"], ["Mortal", "John"]], # Human(John) → Mortal(John)
]
# Fact: Human(John)
fact = ["Human", "John"]

# Query: Mortal(John)?
query = ["Mortal", "John"]
# Apply resolution
if resolution(knowledge_base, query):
    print("Query is resolved: John is Mortal")
else:
    print("Query could not be resolved")

```

Output:

```
Query is resolved: John is Mortal
```

Ex No: 2c

IMPLEMENTATION OF BACKWARD CHAINING

Aim:

To implement backward chaining.

Scenario:

A medical expert system is designed to **diagnose diseases** based on patient symptoms. The system uses **backward chaining** to infer whether a patient has a specific disease by checking rules and known facts.

Procedure:

1. Define the knowledge base with rules (causal relationships).

- "flu": [{"cough", "fever"}] → Flu occurs if both **cough** and **fever** exist.
- "fever": [{"sore_throat"}] → Fever occurs if **sore throat** exists.

2. Define known facts: {sore_throat, cough}.

3. Define the backward chaining function:

- **Check if the goal is in known facts.** If so, return `True`.
- **Check if rules exist for the goal in the knowledge base.**
- **For each rule**, verify **all conditions recursively** using backward chaining.
- If all conditions **can be proven**, return `True`.
- Otherwise, return `False`.

4. Query whether the patient has flu (flu).

5. Execution:

- flu requires cough and fever.
- cough is a **fact** → **True**
- fever **needs** sore_throat.
- sore_throat is a **fact** → **True**
- Since both cough and fever are proven **flu is diagnosed**.

Program:

```

# Knowledge Base (Rules in IF-THEN format)
knowledge_base = {
    "flu": [["cough", "fever"]],
    "fever": [["sore_throat"]],
}

# Known facts
facts = {"sore_throat", "cough"}

# Backward chaining function
def backward_chaining(goal):
    if goal in facts: # If the goal is a known fact, return True
        return True
    if goal in knowledge_base: # If the goal has rules in KB
        for conditions in knowledge_base[goal]: # Check each rule
            if all(backward_chaining(cond) for cond in conditions): # Recursively verify
                return True
    return False # If no rule or fact supports the goal, return False

# Query: Does the patient have flu?
query = "flu"
if backward_chaining(query):
    print(f"The patient is diagnosed with {query}.")
else:
    print(f"The patient does NOT have {query}.")

```

Output:

```
The patient is diagnosed with flu.
```

Ex No: 2d

IMPLEMENTATION OF FORWARD CHAINING

Aim:

To implement forward Chaining.

Scenario:

A **diagnostic expert system** helps determine whether a patient has a disease based on **observed symptoms**. The system uses **forward chaining**, where it starts with known facts (symptoms) and applies rules to infer new facts until it reaches a conclusion (diagnosis).

Procedure:

1. **Initialize a knowledge base** containing **IF-THEN rules**.
2. **Define the initial facts** (observed symptoms or given conditions).
3. **Repeat until no new facts are inferred:**
 - Iterate through each rule in the knowledge base.
 - Check if all conditions (IF part) of a rule exist in the known facts.
 - If true and the conclusion (THEN part) is **not already in facts**, add it to the facts.
 - Mark that a new fact was inferred and continue.
4. **Stop when no new facts are derived** in an iteration.
5. **Check if the final goal or diagnosis is in the inferred facts.**
6. **Output the conclusion** based on derived facts.

Program:

Knowledge Base: Rules in IF-THEN format

```
knowledge_base = [  
    ("cough", "fever"], "flu"),  
    ("sore_throat", "runny_nose"], "cold"),  
    ("sore_throat"], "fever") # Sore throat can lead to fever  
]
```

Given initial facts

```
facts = {"cough", "sore_throat"}
```

Forward Chaining Function

```
def forward_chaining():
```

```
    inferred = True # Keep looping as long as new facts are added
```

```

while inferred:
    inferred = False # Stop if no new fact is added in an iteration

    for conditions, conclusion in knowledge_base:
        if all(condition in facts for condition in conditions) and conclusion not in facts:
            facts.add(conclusion) # Add the inferred fact
            inferred = True # Mark that we inferred a new fact

# Run forward chaining
forward_chaining()

# Check if flu or cold is inferred
if "flu" in facts:
    print("The patient is diagnosed with flu.")
elif "cold" in facts:
    print("The patient is diagnosed with cold.")
else:
    print("No conclusive diagnosis could be made.")

```

Output:

```
The patient is diagnosed with flu.
```

Ex No: 3a

IMPLEMENTATION OF BLOCKS WORLD PROGRAM

Aim:

To implement Blocks World Program.

Scenario:

A **robotic arm** in a warehouse is programmed to **rearrange blocks** according to a given goal state. The **Blocks World problem** involves moving blocks from an initial configuration to a desired goal configuration while following specific constraints.

A robotic system is given an **initial state** and a **goal state**:

Initial State:

A is on B

B is on table

C is on table

Goal State

B is on C

A is on B

C is on table

Procedure:

1. **Initialize the world** with an initial state of blocks.
2. **Define the goal state** that needs to be achieved.
3. **Check if the current state matches the goal state:**
 - If **yes**, stop the execution.
 - If **no**, continue planning moves.
4. **For each block in the goal state:**
 - If the block is not in its desired position, **move it** to the correct place.
 - Print the move action.
 - Update the current state after each move.
5. **Repeat until the goal state is reached.**
6. **Print the final arrangement of blocks** when the goal state is met.

Program:

```

class BlocksWorld:
    def __init__(self):
        self.state = {
            "A": "B", # A is on B
            "B": "table", # B is on table
            "C": "table" # C is on table
        }
        self.goal = {
            "A": "B",
            "B": "C",
            "C": "table"
        }

    def is_goal_state(self):
        return self.state == self.goal

    def move(self, block, destination):
        if block in self.state and self.state[block] != destination:
            print(f'Moving {block} from {self.state[block]} to {destination}')
            self.state[block] = destination

    def plan_moves(self):
        print("\nInitial State:", self.state)
        while not self.is_goal_state():
            for block, target in self.goal.items():
                if self.state[block] != target:
                    self.move(block, target)

        print("\nFinal Goal State Reached:", self.state)

# Run the Blocks World Solver
bw = BlocksWorld()
bw.plan_moves()

```

Output:

Initial State: {'A': 'B', 'B': 'table', 'C': 'table'}

Moving B from table to C

Moving A from B to B

Moving C from table to table

Final Goal State Reached: {'A': 'B', 'B': 'C', 'C': 'table'}

Ex No: 3b

IMPLEMENTATION OF A FUZZY INFERENCE SYSTEM

Aim:

To implement Fuzzy Inference System.

Scenario:

A company wants to automate **employee performance evaluation** based on two factors:

1. **Work Experience (Years)**
2. **Project Success Rate (%)**

Using **Fuzzy Logic**, we classify employee performance as **Poor, Average, or Excellent**, which helps determine bonuses or promotions.

The system follows these rules:

- **If experience is low AND success rate is low → Performance is Poor.**
- **If experience is medium OR success rate is medium → Performance is Average.**
- **If experience is high AND success rate is high → Performance is Excellent.**

Procedure:

1. Define Input Variables:

- Experience (0 to 20 years)
- Success Rate (0 to 100%)

2. Define Output Variable:

- Performance Score (0 to 100%)

3. Create Fuzzy Membership Functions for Experience, Success Rate, and Performance:

- Low, Medium, High (for input variables)
- Poor, Average, Excellent (for output variable)

4. Define Fuzzy Rules:

- IF experience is low AND success rate is low → THEN performance is poor.
- IF experience is medium OR success rate is medium → THEN performance is average.
- IF experience is high AND success rate is high → THEN performance is excellent.

5. **Build the Fuzzy Inference System (FIS)** using control rules.
6. **Provide Input Values:**
 - Example: Experience = 12 years, Success Rate = 70%
7. **Perform Fuzzy Computation** to determine the final performance score.
8. **Output the Performance Score** based on fuzzy logic inference.

Program:

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Define fuzzy variables
experience = ctrl.Antecedent(np.arange(0, 21, 1), 'experience')
success_rate = ctrl.Antecedent(np.arange(0, 101, 1), 'success_rate')
performance = ctrl.Consequent(np.arange(0, 101, 1), 'performance')

# Define fuzzy membership functions
experience['low'] = fuzz.trimf(experience.universe, [0, 0, 10])
experience['medium'] = fuzz.trimf(experience.universe, [5, 10, 15])
experience['high'] = fuzz.trimf(experience.universe, [10, 20, 20])

success_rate['low'] = fuzz.trimf(success_rate.universe, [0, 0, 50])
success_rate['medium'] = fuzz.trimf(success_rate.universe, [25, 50, 75])
success_rate['high'] = fuzz.trimf(success_rate.universe, [50, 100, 100])

performance['poor'] = fuzz.trimf(performance.universe, [0, 0, 50])
performance['average'] = fuzz.trimf(performance.universe, [25, 50, 75])
performance['excellent'] = fuzz.trimf(performance.universe, [50, 100, 100])
```

```
# Define fuzzy rules
rule1 = ctrl.Rule(experience['low'] & success_rate['low'], performance['poor'])
rule2 = ctrl.Rule(experience['medium'] | success_rate['medium'], performance['average'])
rule3 = ctrl.Rule(experience['high'] & success_rate['high'], performance['excellent'])

# Create FIS control system
performance_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
performance_sim = ctrl.ControlSystemSimulation(performance_ctrl)

# Provide input values
performance_sim.input['experience'] = 12 # Example: 12 years of experience
performance_sim.input['success_rate'] = 70 # Example: 70% success rate

# Compute fuzzy inference
performance_sim.compute()

# Print the output
print(f'Predicted Performance Score: {performance_sim.output['performance']:.2f}')
```

Output

```
Predicted Performance Score: 67.85
```

VIVA QUESTIONS WITH ANSWERS

1. What is Artificial Intelligence (AI)?

Answer: Artificial Intelligence (AI) refers to the simulation of human intelligence in machines that are programmed to think and act like humans.

2. What is an intelligent agent?

Answer: An intelligent agent is a system that perceives its environment and takes actions to achieve its goals autonomously.

3. What are the types of environments an agent can operate in?

Answer: Agents can operate in environments that are deterministic, observable, static, and dynamic, depending on the properties of the environment.

4. What is the difference between a reflex agent and a goal-based agent?

Answer: A reflex agent acts based on current perceptions using predefined rules, while a goal-based agent evaluates actions to achieve specific goals.

5. What is a utility-based agent?

Answer: A utility-based agent selects actions that maximize its expected utility or satisfaction based on available options.

6. What is the state space in AI?

Answer: The state space is the set of all possible states that can be reached during the process of solving a problem.

7. What are the components of a production system?

Answer: A production system consists of rules, a knowledge base, an inference engine, and working memory.

8. What is the difference between a well-defined problem and an ill-defined problem?

Answer: A well-defined problem has clear objectives and constraints, while an ill-defined problem lacks clear goals and solutions.

9. What are the key issues in designing search programs?

Answer: Key issues include time complexity, space complexity, optimality, completeness, and efficiency in solving a problem.

10. What is the role of fuzziness in fuzzy logic systems?

Answer: Fuzziness in fuzzy logic allows for reasoning and decision-making with imprecise, uncertain, or approximate information.

11. What is a problem-solving agent?

Answer: A problem-solving agent is an intelligent agent that takes actions to reach a desired goal from an initial state through a search process.

12. What is the Breadth-First Search (BFS) strategy?

Answer: BFS is a search strategy that explores all possible nodes at the present depth level before moving on to nodes at the next depth level.

13. What is Depth-First Search (DFS)?

Answer: DFS is a search strategy that explores as far down a branch of the tree as possible before backtracking to explore other branches.

14. What is Depth-Limited Search?

Answer: Depth-Limited Search is a variation of DFS that limits the depth of search to a pre-defined depth level to prevent infinite loops.

15. What is Bidirectional Search?

Answer: Bidirectional Search simultaneously searches from both the initial state and the goal state, meeting in the middle to find the solution more efficiently.

16. What is Greedy Best-First Search?

Answer: Greedy Best-First Search selects the node that appears to be closest to the goal based on a heuristic function.

17. What is the A Search algorithm?*

Answer: A* Search is a heuristic search algorithm that evaluates nodes based on both the cost to reach the node and the estimated cost to the goal, providing optimal solutions.

18. What is AO Search?*

Answer: AO* Search is an algorithm used for solving problems in a Directed Acyclic Graph (DAG) by combining the concepts of A* and AND/OR graphs.

19. What is Hill Climbing Search?

Answer: Hill Climbing Search is a local search algorithm that continuously moves towards the neighbor with the highest value, optimizing a specific objective function.

20. What is Simulated Annealing?

Answer: Simulated Annealing is an optimization algorithm that explores the solution space by allowing occasional moves to worse solutions to escape local optima, inspired by the annealing process in metallurgy.

21. What is local search for constraint satisfaction problems (CSPs)?

Answer: Local search for CSPs involves iteratively improving a candidate solution by making small changes to reduce constraint violations, aiming to find a solution that satisfies all constraints.

22. What is adversarial search in AI?

Answer: Adversarial search is a process where agents compete against each other, and each agent aims to maximize its own benefit while minimizing the opponent's benefit, typically used in games like chess.

23. How are optimal decisions made in games?

Answer: Optimal decisions in games are made by evaluating all possible moves and selecting the one that maximizes a player's chances of winning, considering both the current state and possible future moves.

24. What is the Min-Max search procedure?

Answer: The Min-Max search procedure is an algorithm used in game theory to recursively determine the best move by assuming that the opponent will also play optimally, minimizing the maximizing player's benefit.

25. What is Alpha-Beta pruning in adversarial search?

Answer: Alpha-Beta pruning is an optimization technique for Min-Max search that eliminates branches of the game tree that will not affect the final decision, improving search efficiency.

26. . What is knowledge representation in AI?

Answer: Knowledge representation in AI is the process of encoding information about the world in a form that a computer system can use to solve complex tasks like reasoning and problem-solving.

27. What is rule-based knowledge representation?

Answer: Rule-based knowledge representation uses a set of "if-then" rules to encode knowledge, where each rule represents a relationship or condition about the world.

28. What is the difference between procedural and declarative knowledge?

Answer: Procedural knowledge describes how to perform tasks or actions, while declarative knowledge represents facts or information about the world.

29. What is logic programming in AI?

Answer: Logic programming is a programming paradigm where programs are written as sets of logical statements, and computation is performed through logical inference.

30. What is the difference between forward and backward reasoning?

Answer: Forward reasoning starts from known facts and applies rules to reach a conclusion, while backward reasoning starts from a goal and works backward to find supporting facts.

31. What is Bayesian Probability and Bayes' Theorem?

Answer: Bayesian Probability uses Bayes' Theorem to update the probability of a hypothesis based on new evidence, providing a method to calculate conditional probabilities.

32. What are Certainty Factors in Rule-Based Systems?

Answer: Certainty factors quantify the degree of belief or confidence in a conclusion drawn from a rule-based system, allowing reasoning under uncertainty.

33. What are Bayesian Networks?

Answer: Bayesian Networks are graphical models that represent probabilistic relationships between variables, where nodes represent variables and edges represent conditional dependencies.

34. What is the Dempster-Shafer Theory?

Answer: The Dempster-Shafer Theory is a mathematical framework for reasoning with uncertainty, allowing the combination of evidence from different sources to assign belief functions to hypotheses.

35. What is a Markov Decision Process (MDP)?

Answer: A Markov Decision Process is a mathematical model for decision-making where an agent interacts with an environment in discrete time steps, with outcomes determined by both the agent's actions and probabilistic transitions.