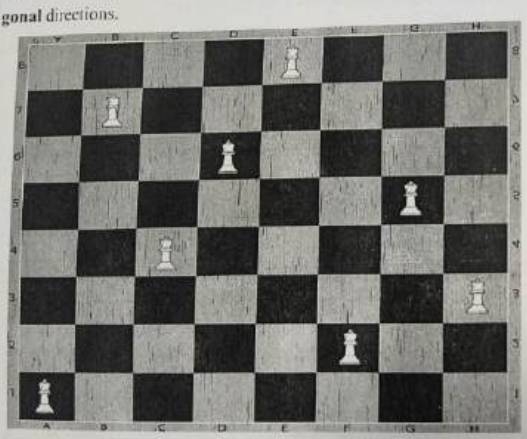
In-Lab Discussion:

Aim:

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

Case Scenario:

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

Output:

```
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
1 0 0 0 0 0 0 0
```

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 . . . .
```

Post – Lab Discussion:

Real Life Application:

- **Timetabling and Scheduling:** Avoiding conflicts in assigning resources like classrooms, employees, or transportation.

- **Resource Allocation:** Distributing ... without interference.
- **Logistics Optimization:** Planning routes and layouts to prevent conflicts ... maximize efficiency.
- **Game AI Development:** Implementing intelligent agents that can make non-conflicting moves (e.g., in chess or similar strategy games).
- **Robotics Path Planning:** Navigating robots through environments while ... obstacles.
- **Automated Reasoning Systems:** Solving logical puzzles and proving theorems ... exploring non-conflicting states.
- **VLSI Design:** Placing components on integrated circuits without overlaps ... electrical conflicts.
- **Software Engineering (Algorithm Design & Scalability):** Understanding ... addressing the challenges of solving computationally complex problems with increasing constraints.

How many solutions are there for 8 queens on 18 to 8 board?

Explanation: There are total of 12 fundamental solutions to the eight queen puzzle ... removing the symmetrical solutions due to rotation. For 8*8 chess board with 8 queens ... are total of 92 solutions for the puzzle.

Case-Based Discussion:

Write a Python program that solves the **N-Queens problem** using the **Backtracking algorithm**.

Result :-

The Implement of Eight Queens problems is executed Successfully and Verified

Ex No: 1b

IMPLEMENTATION OF DEPTH FIRST SEARCH

Pre-Lab Discussion:

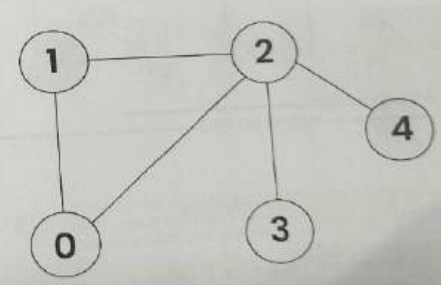
Depth First Search or DFS for a Graph – Python

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a Boolean visited array. A graph can have more than one DFS traversal.

Example:

Note : There can be multiple DFS traversals of a graph according to the order in which we pick adjacent vertices. Here we pick vertices as per the insertion order.

Input: adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]



Output: 1 0 2 3 4

Explanation: The source vertex s is 1. We visit it first, then we visit an adjacent.

Start at 1: Mark as visited. Output: 1

Move to 0: Mark as visited. Output: 0 (backtrack to 1)

Move to 2: Mark as visited. Output: 2 (backtrack to 0)

- 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']
```

Post – Lab Discussion:

Real-time applications:

While DFS can be computationally expensive in very large or complex graphs, it can be effective in real-time scenarios where the size of the problem space is manageable and the goal is to find a solution quickly rather than necessarily the optimal one. For example, in a game-playing AI, DFS can be used to explore possible moves and find a good strategy within a time limit.

Applications of DFS in AI

- **Maze generation:** The Maze generation is comprised of designing a layout of passages and walls within a maze. This maze generation makes use of a randomized approach of the Depth-first search algorithm because it leverages the recursive method and stack. For instance, assume that the space is a large grid of cells where each cell holds the four walls. The DFS performs by selecting any random neighbor at first that has not been visited. It removes the wall between the two cells that are not connected and then it adds the new cell to the stack. This process continues until there is no more solution can be generated, resulting in a complete maze.
- **Puzzle-solving:** Puzzle-solving including Japanese nonograms can employ Depth-first search as a method for systematically exploring possible solutions. In Japanese nonograms, DFS is utilized to explore different combinations of filled and empty cells in the grid.

- **Pathfinding in robotics:** DFS can be employed for pathfinding in robotics, especially in scenarios where simplicity, memory efficiency, and adaptability are important considerations.

Case-Based Discussion:

Write a python code for Tic-Tac-Toe game using DFS.

Result:-

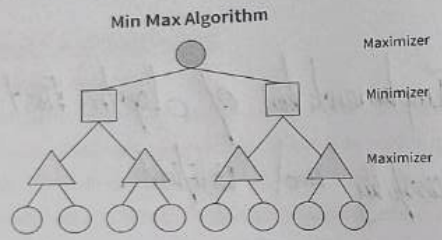
The Implementation of depth first Search is executed successfully and verified.

Ex No: 1c

IMPLEMENTATION OF MINIMAX algorithm

Pre-Lab Discussion:

The minimax algorithm in game theory helps the players in two-player games decide their best move. It is crucial to assume that the other player is also making the best move while determining the best course of action for the present player. Each player attempts to reduce the maximum loss that they can suffer in the game. This article will cover the minimax algorithm's concept, its working, its properties, and other relevant ideas.



Working of Min-Max Process in AI

Min-Max algorithm involves two players: the maximizer and the minimizer, each aiming to optimize their own outcomes.

Players Involved

Maximizing Player (Max):

- Aims to maximize their score or utility value.
- Chooses the move that leads to the highest possible utility value, assuming the opponent will play optimally.

Minimizing Player (Min):

- Aims to minimize the maximizer's score or utility value.
- Selects the move that results in the lowest possible utility value for the maximizer, assuming the opponent will play optimally.

The interplay between these two players is central to the Min-Max algorithm, as each player attempts to outthink and counter the other's strategies.

Steps involved in the Mini-Max Algorithm

The Min-Max algorithm involves several key steps, executed recursively until the optimal move is determined. Here is a step-by-step breakdown:

Step 1: Generate the Game Tree

- **Objective**: Create a tree structure representing all possible moves from the current game state.
- **Details**: Each node represents a game state, and each edge represents a possible move.

Step 2: Evaluate Terminal States

- **Objective**: Assign utility values to the terminal nodes of the game tree.
- **Details**: These values represent the outcome of the game (win, lose, or draw).

Step 3: Propagate Utility Values Upwards

- **Objective**: Starting from the terminal nodes, propagate the utility values upwards through the tree.
- **Details**: For each non-terminal node:
  - If it's the maximizing player's turn, select the maximum value from the child nodes.
  - If it's the minimizing player's turn, select the minimum value from the child nodes.

Step 4: Select Optimal Move

- **Objective**: At the root of the game tree, the maximizing player selects the move that leads to the highest utility value.

Min-Max Formula

The Min-Max value of a node in the game tree is calculated using the following recursive formulas:

1. Maximizing Player's Turn:

$\text{Max}(s)=\max_{a \in A(s)}\text{Min}(\text{Result}(s,a))\text{Max}(s)=\max_{a \in A(s)}\text{Min}(\text{Result}(s,a))$

Here:

- $\text{Max}(s)\text{Max}(s)$is the maximum value the maximizing player can achieve from state s.
- $A(s)A(s)$ is the set of all possible actions from state s.

3. Create `isMovesLeft(board)` ... moves ...
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, ... call `minimax(board, False)`, undo move, update `best` with maximum ... return `best`.
- If `isMax` is `False` (Human's turn), initialize `best = +∞`, loop through empty ... place `O`, call `minimax(board, True)`, undo move, update `best` with minimum ... 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 ... `bestMove` if a better move is found.
- Return `bestMove`.

6. Implement `printBoard(board)` to display board state using `"X"`, `"O"`, and `"."` for ... 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
```

Post-Lab Discussion:

Real Time Applications of the MinMax Algorithm:

Game AI

Game AI uses the MiniMax algorithm quite frequently. Game designers use the algorithm to build AI opponents that are capable of playing poker, tic-tac-toe, and chess at a high level. Even in complex games with enormous search spaces, the AI can make the best decisions by using the MinMax algorithm.

Decision-Making

The MinMax algorithm can also be utilized in decision-making processes, such as **financial planning and resource allocation**. Decision-makers can make better choices and limit losses by using this algorithm.

Auctions

The MinMax algorithm can be used in auctions to determine the optimal bid for a
The algorithm can help a bidder make an informed decision and minimize their
evaluating the other bidders' potential moves and their maximum possible gain.

Negotiations

The Min Max algorithm can also be used to decide on a negotiator's best course of
analyzing the various moves of the opposite side and their maximum gain, the
assist a negotiator in making an informed choice and maximizing their gains.

Limitation of the minimax Algorithm

The following are some of the drawbacks of the Min-Max algorithm:

- It presumes that the adversary likewise makes the best movements, which
the case in actual play. Players might make blunders or employ suboptimal
real-world games. The MinMax algorithm may take a longer time to
player's best move in such circumstances.
- The games with a wide search space should not use the MinMax algorithm.
Evaluating every move in such games could take a while, and the memory
might become excessively large.
- The MinMax algorithm does not consider the probability of certain events
occurring. The outcome of some games, like poker, is determined by the
that specific things will happen, such as the allocation of cards. The MinMax
algorithm might not be appropriate in these games.

Future Developments of the MinMax Algorithm

The Min Max algorithm has been a cornerstone of game theory and artificial
many years. However, there are still several areas where the algorithm could be
improved. The following are some of the potential MinMax algorithms advances for
future:

Multi-Player Games

The MinMax algorithm was initially created for two-player games. Yet, several games,
like **bridge, poker, and go**, have more than two participants. The MinMax algorithm

being extended to multi-player games by researchers, who may employ game-theoretic models like the Shapley value.

Deep Reinforcement Learning

For representing the value function in the MinMax method, deep reinforcement learning, a version of reinforcement learning, uses deep neural networks. The system can learn more complicated strategies and play games at a more significant position by utilizing deep reinforcement learning.

Uncertainty

The Min Max algorithm assumes that both players know the game's rules and can weigh all potential plays. However, the opponent's actions or the game's condition are unpredictable in many real-world situations. Researchers are exploring ways to extend the MinMax algorithm to handle uncertainty, such as by using Bayesian models or Monte Carlo simulations. These MinMax algorithm improvements can potentially broaden the method's applications and enhance its functionality. The MinMax algorithm is anticipated to continue to be a key component of strategic planning and decision-making as game theory and artificial intelligence develop.

Case-Based Discussion:

Write a python program for poker game using minmax algorithm.

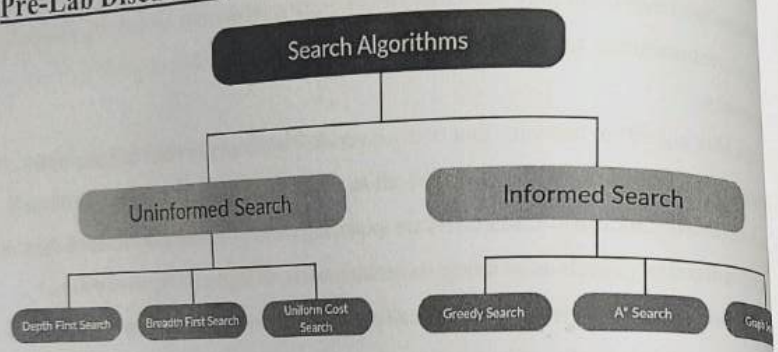
Result:

The Implementation of MINMAX algorithm is executed successfully and verified

Ex No: 1d

IMPLEMENTATION OF A* SEARCH ALGORITHM

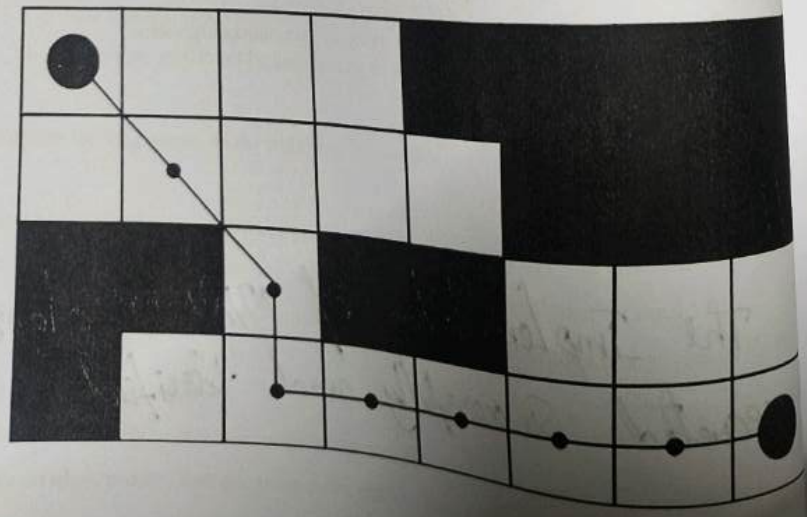
Pre-Lab Discussion:



A* Search:

To approximate the shortest path in real-life situations, like- in maps, games where there be many hindrances.

We can consider a 2D Grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green below)



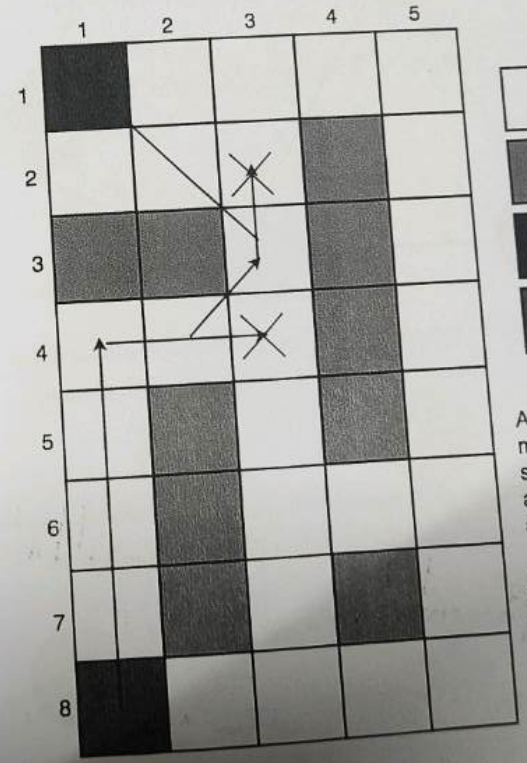
What is A* Search Algorithm?

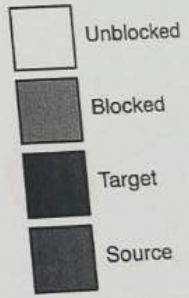
A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).





A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

- Add the current node ...
- Generate new valid moves (up, down, left, right) ensuring they are within b...
  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)]
```

CAT-2 QUESTION PATTERN .....**(I think)**

1.a) section of solids (15 M)

Or

1.b) section of solids (15 M)

2.a) Development of solids (15 M)

Or

2.b) Development of solids (15 M)

3.a) Isometric projection (unit 4) (15 M)

Or

3.b) Isometric projection (unit 4) (15 M)

4.a) Freehand sketching (multiple views /pictorial view) (15 M)

Or

4.b) Freehand sketching (multiple views /pictorial view) (15 M)

5.a) perspective projection (15M)

Or

5.b) Perspective projection (15 M)

Total mark: 75, time: 2 hrs.

Post-Lab Discussion:

REAL-WORLD APPLICATIONS:

- **Navigation and Mapping Systems:** This is probably the most recognizable application. Think of your car's GPS, Google Maps, or any other route-pla... A* is often used to find the shortest or most efficient path between two poi... considering factors like distance, traffic, road closures, and even speed limi... intelligently explores potential routes, prioritizing those that seem most pro...
- **Robotics and Autonomous Vehicles:** For a robot to navigate a complex en... or an autonomous car to plan its path, it needs to find the best way to get fro... to point B while avoiding obstacles. A* helps these systems make intelligent decisions about movement and trajectory planning in real-time.
- **Game AI:** In video games, especially strategy or role-playing games, non-pl... characters (NPCs) need to move intelligently around the game world. A* is frequently used to enable characters to find their way through intricate level... obstacles, and chase or evade the player in a realistic manner.
- **Logistics and Supply Chain Management:** Companies like Amazon or logi... providers use pathfinding algorithms to optimize delivery routes for their veh... can be adapted to consider multiple delivery locations, time windows, vehicle capacities, and other constraints to find the most cost-effective and efficient d... schedules.
- **Warehouse Management:** Within large warehouses, robots and automated gu... vehicles (AGVs) are used to move goods around. A* helps in planning the op... paths for these machines to pick up and deliver items, minimizing travel time ... maximizing efficiency.
- **Network Routing:** In computer networks, including the internet, data packets ... find the best path to travel from a source to a destination. While the actual rou... protocols are more complex, the fundamental idea of finding the shortest or m... efficient path, as in A*, is a key concept.
- **Resource Allocation and Task Scheduling:** In certain applications, like allo... tasks to processors in a multi-core system or scheduling jobs in a manufactur... plant, algorithms inspired by A* can be used to find the most efficient way to allocate resources or schedule tasks to minimize completion time or costs.
- **Medical Diagnosis and Treatment Planning:** While perhaps less direct, search algorithms like A* can be adapted and used in AI systems for medical diagnosis to explore the "path" of potential diagnoses based on symptoms and test results. Similarly, in treatment planning, it could help find the optimal sequence of treatments.

Case-Based Discussion:

"Escape the Monster" - A* for Evading a Simple Opponent:

- **Topic:** Implementing A* for a player character to find the shortest path to an exit while avoiding a stationary "monster" on a grid.
- **Lab Task:** Students implement A* for the player. The monster's position is fixed. The goal is to find the shortest path to the exit that doesn't collide with the monster. This adds a simple game-like element.

Result:-

The Implementation of A* Search Algorithm is executed successfulled and verified.

Ex No: 2

IMPLEMENTATION OF DECISION MAKING AND KNOWLE REPRESENTATION

Pre-Lab Discussion:

What Is Prolog?

Prolog is a declarative and logic programming language designed for developing logic AI applications. Developers can set rules and facts around a problem, and then Prolog interpreter will use that information to automatically infer solutions.

Prolog Program Basics to Know

In Prolog, programs are made up of two main components: facts and rules. Facts are statements that are assumed to be true, such as "John is a man" or "the capital of France Paris." Rules are logical statements that describe the relationships between different fac such as "If John is a man and Mary is a woman, then John is not Mary."

Prolog programs are written using a syntax that is similar to natural language. For exam simple Prolog program might look like this:

```
man(john).
woman(mary).
capital_of(france, paris).

not_same(X,Y) :- man(X), woman(Y).
```

In this example, the first three lines are facts, while the fourth line is a rule. The rule use the not_same/2 predicate to state that if X is a man and Y is a woman, then X is not the s as Y.

How Is Prolog Different From Other Programming Languages?

Prolog is a declarative programming language, while common like Python or JavaScript are imperative. A de should do, and focuses on th

One of the key features of Prolog is its ability to handle uncertain or incomplete information. In Prolog, a programmer can specify a set of rules and facts that are known to be true, but they can also specify rules and facts that might be true or false.

The Prolog interpreter will then use those rules and facts to automatically reason about the problem domain and find solutions that are most likely to be correct, given the available information.

How to Use Prolog

One way to use Prolog is to define a set of rules that describe the relationships between different objects or concepts in your problem domain. For example, you might define rules that specify that certain objects are bigger than others, or that some objects are the same color. Then, you can use Prolog to ask questions about these objects and their relationships, and the interpreter will use your rules to deduce the answers.

To use Prolog, you will need to have a Prolog interpreter installed on your computer. There are several different Prolog interpreters available, including SWI-Prolog, GNU Prolog and B-Prolog. Once you've installed an interpreter, you can start writing Prolog programs using a text editor and then run them using the interpreter.

How Prolog Syntax Works

There is no single "syntax" for Prolog, as the language allows for a wide range of different programming styles and approaches. However, here are some basic elements of Prolog syntax that are commonly used:

- **Facts** are statements that are assumed to be true. In Prolog, facts are written using a predicate name followed by a list of arguments enclosed in parentheses. For example: man(john).
- **Rules** are logical statements that describe the relationships between different facts. In Prolog, rules are written using the predicate name followed by a list of arguments enclosed in parentheses, followed by a colon and a hyphen (:-) and the body of the rule. For example: happy(X) :- likes(X, pizza).
- **Variables** are used to represent values that can change or be determined by the interpreter. In Prolog, variables are written using a name that begins with an

characteristics. These taxo... kingdoms, phyla... orders, families, genera, and species. This statement represents structural k... because it describes the hierarchical structure of the taxonomic classifica... used in biology. It acknowledges that there are specific levels of organizat... this system and that each level has its unique characteristics and relationsh... levels.

The Relation Between Knowledge and Intelligence

Knowledge and intelligence are related but distinct concepts. Knowledge refers to ... information, skills, and understanding that an individual has acquired through learn... experience. In contrast, intelligence refers to the ability to think abstractly, reason, ... quickly, solve problems, and adapt to new situations.

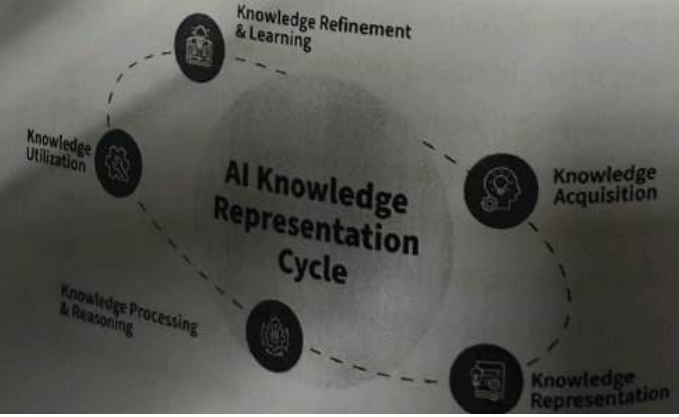
In the context of AI, knowledge, and intelligence are also distinct but interrelated co... AI systems can be designed to acquire knowledge through machine learning or exp... systems. Still, the ability to reason, learn, and adapt to new situations requires a mor... general intelligence that is beyond most AI systems' capabilities.

An agent can only act accurately on some input when it has some knowledge or experience about that input.

Nonetheless, using knowledge-based systems and other AI techniques can help enha... intelligence of machines and enable them to perform a wide range of tasks.

The AI Knowledge Cycle

The AI Knowledge Cycle represents the continuous process through which AI syste... acquire, process, utilize, and refine knowledge.



How Agents Use Knowledge to Make Decisions

Imagine a robot that cleans houses. It uses all these types of knowledge to do its job:

[[artifact_table]] Example of knowledge types used by a house-cleaning robot [[/artifact_table]]

- It uses structural knowledge to understand the layout of a house.
- Procedural knowledge helps it know how to clean different surfaces.
- Heuristic knowledge might tell it to clean high-traffic areas more often.
- Meta-knowledge helps it know when to ask for help if it encounters something new.

In-Lab Discussion:

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 ;

Post – Lab Discussion:

Real-World Example: Self-Driving Cars

Self-driving cars are a great example of intelligent agents using knowledge representation. They need to make quick decisions to drive safely. Here's how they use different types of knowledge:

- **Structural:** Maps and road layouts
- **Procedural:** Rules of the road and how to operate the car

- **Heuristic:** Quick judgments about other drivers' behavior
- **Meta:** Understanding the limits of its sensors in bad weather

By combining these types of knowledge, self-driving cars can navigate complex to make split-second decisions to keep passengers safe.

Challenges in Implementing Knowledge Representation

Creating smart agents isn't easy. Some big challenges are:

- Making sure the knowledge is accurate and up-to-date
- Helping agents understand context and nuance
- Balancing quick decisions with thorough analysis
- Dealing with new situations the agent hasn't seen before

Researchers are always working on better ways to represent knowledge in agents. A improve, we'll see smarter and more helpful AI in our daily lives.

Case-Based Discussion:

Self-driving cars are excellent examples of intelligent agents utilizing kno representation. One critical module in such systems is the **lane change de making** component. This module determines whether it's legal and safe for vehicle to change lanes based on current road conditions and traffic rules. **Write a Pr program for that Module.**

Result:-

The Implementation of Decision Making and Knowledge Representation is executed Successfully Verified.

Ex No: 2b

IMPLEMENTATION OF UNIFICATION AND RESOLUTION ALGORITHM

Pre-Lab Discussion:

Unification in AI is a core concept in logic and automated reasoning that enables the matching of logical expressions by identifying and substituting variables. It plays a crucial role in theorem proving, inference systems, and symbolic processing by ensuring consistency in logical statements. Unification is widely used in first-order logic, Prolog programming, and natural language processing to enhance rule-based reasoning.

What is Unification in AI?

Unification in AI is the process of making two logical expressions identical by determining a suitable substitution of variables. It is a key operation in first-order logic and is widely used in automated reasoning, inference engines, and logic programming to resolve logical statements systematically.

In AI, unification plays an essential role in theorem proving, where it helps match hypotheses with conclusions in logical deductions. In logic programming languages like Prolog, unification enables the system to match rules and facts to queries, allowing efficient pattern matching and rule evaluation.

Example of Unification in Predicate Logic

Consider two logical expressions in predicate logic:

1. Parent(X, Mary)
2. Parent(John, Mary)

To unify these expressions, we find a substitution that makes them identical. Here, substituting **X = John** results in:

Parent(John, Mary) = Parent(John, Mary)

Since the expressions are now identical, unification is successful. This process allows AI systems to **infer new knowledge and establish logical relationships**, making it fundamental in knowledge-based reasoning and automated decision-making.

Importance of Unification in AI

- Otherwise, retu

2. **Define the variable unification function (**
   - If the variable already exists in the substitution set, apply unification recursi
   - 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 th** **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 resol

Output:

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

Post-Lab Discussion:

Future Directions in Unification

1. **Artificial General Intelligence (AGI):** The ultimate goal of unification is A achieve AGI—a system with human-like reasoning and problem-solving abi across all domains.
2. **Interdisciplinary Collaboration:** The future of unified AI depends on collab across fields, including neuroscience, cognitive science, and computer scienc

Federated and Distributed AI Systems: Unification may also involve connecting m AI agents through distributed networks, leveraging federated learning to create a decentralised, collaborative AI ecosystem.

Applications of the Resolution Algorithm in AI

The Resolution Algorithm is widely used in the following AI applications:

1. **Automated Theorem Proving** : It helps computers to automatically prove mathematical theorems or verify logical arguments without human interventio
2. **Knowledge Representation** : In AI systems knowledge is represented as logic statements. The Resolution Algorithm allow these systems to reason about thei knowledge effectively.
3. **Problem Solving** : Many real-world problems can be framed as logical puzzles example scheduling tasks, diagnosing faults in systems or planning routes can benefit from logical reasoning using resolution.
4. **Foundation for Advanced Techniques** : The Resolution Algorithm forms the for more advanced AI techniques such as **SAT solvers** which is used to solve Boolean satisfiability problems and **logic programming languages.**

Case-Based Discussion:

Try to write the above program in prolog and give the difference between the implementation.

Result:

Implementation of Unification and Resolution Algorithm is executed successfully and verified.

Ex No: 2c

IMPLEMENTATION OF BACKWARD CHAINING

Pre-Lab Discussion:

Horn Clause and Definite clause:

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the **first-order definite clause**.

Definite clause: A clause which is a disjunction of literals with **exactly one positive literal** is known as a definite clause or strict horn clause.

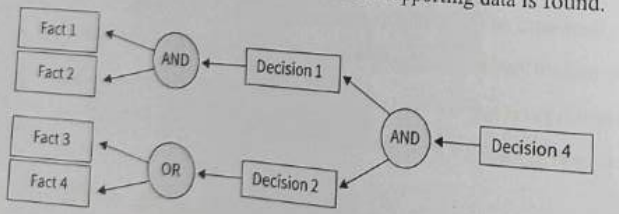
Horn clause: A clause which is a disjunction of literals **with at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: $(\neg p \vee \neg q \vee k)$. It has only one positive literal k.

It is equivalent to $p \wedge q \rightarrow k$.

What is Backward Chaining?

Backward chaining is a **goal-driven** reasoning strategy used in AI. It starts with a goal or hypothesis and works backward to determine if the available facts support the goal. The process continues by recursively breaking down the goal into smaller sub-goals until either all facts are verified or no more supporting data is found.



Properties of Backward Chaining:

- **Goal-Driven:** Reasoning begins with a desired goal and searches for evidence to support it.
- **Top-Down Approach:** The system starts from the goal and works back to find relevant facts.

- **Depth-First Search Strategy:** The inference engine follows a path deeply before exploring other possibilities, prioritizing each goal or sub-goal in sequence.
- **Possibility of Infinite Loops:** If not handled properly, backward chaining may get stuck in loops while looking for evidence to support the goal.

Example of Backward Chaining 1:

Let's consider a medical diagnosis system where the goal is to determine if a patient has the flu. The system starts with known facts:

1. **Fact 1:** The patient has a fever.
2. **Fact 2:** The patient has a sore throat.
3. **Rule:** If the patient has a fever and sore throat, they might have the flu.

The system applies the rule, leading to the conclusion that the patient might have the flu.

Solution:

1. **Goal:** Does the patient have the flu?
2. **Rule:** If the patient has a fever and sore throat, they might have the flu.
3. **Sub-goals:**
   - Verify if the patient has a fever.
   - Verify if the patient has a sore throat.

The system works backward from the goal (flu) and checks if the patient's symptoms (fever and sore throat) match. If all sub-goals are verified, the system concludes that the patient likely has the flu.

How Backward Chaining Works

1. **Start with a Goal**: The inference engine begins with the goal or hypothesis it wants to prove.
2. **Identify Rules**: It looks for rules that could conclude the goal.
3. **Check Conditions**: For each rule, it checks if the conditions are met, which may involve proving additional sub-goals.
4. **Recursive Process**: This process is recursive, working backward through the rule set until the initial facts are reached or the goal is deemed unattainable.

Example of Backward Chaining 2:

In a troubleshooting system for network issues:

- **Goal**: Determine why the network is down.
- **Rule**: If the router is malfunctioning, the network will be down.

The system starts with the goal (network down) and works backward to check if the rout malfunctioning, verifying the necessary conditions to confirm the hypothesis.

In-Lab Discussion:

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 checkin 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.
```

Post-Lab Discussion:

Advantages of Backward Chaining

1. **Goal-Oriented**: It is efficient for goal-specific tasks as it only generates the facts needed to achieve the goal.
2. **Resource Efficient**: It typically requires less memory, as it focuses on specific goals rather than exploring all possible inferences.
3. **Interactive**: It is well-suited for interactive applications where the system needs to answer specific queries or solve particular problems.
4. **Suitable for Diagnostic Systems**: It is particularly effective in diagnostic systems where the goal is to determine the cause of a problem based on symptoms.

Disadvantages of Backward Chaining

1. **Complex Implementation**: It can be more complex to implement, requiring sophisticated strategies to manage the recursive nature of the inference process.
2. **Requires Known Goals**: It requires predefined goals, which may not always be feasible in dynamic environments where the goals are not known in advance.
3. **Inefficiency with Multiple Goals**: If multiple goals need to be achieved, backward chaining may need to be repeated for each goal, potentially leading to inefficiencies.
4. **Difficulty with Large Rule Sets**: As the number of rules increases, managing the backward chaining process can become increasingly complex.

Case-Based Discussion:

Try to write a python program for troubleshooting system for network issues using backward chaining

Result:

Implementation of Backward Chaining is executed Successfully and Verified.

Ex No: 2d

IMPLEMENTION OF FORWARD CHAINING

Pre-Lab Discussion:

Horn Clause and Definite clause:

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the **first-order definite clause**.

Definite clause: A clause which is a disjunction of literals with **exactly one positive literal** is known as a definite clause or strict horn clause.

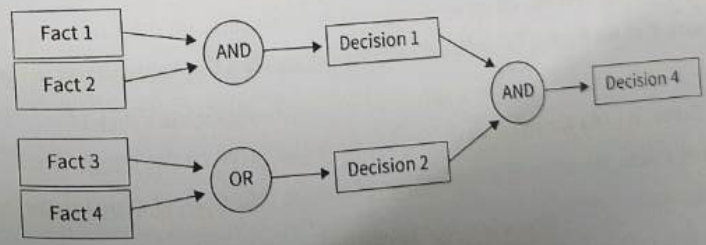
Horn clause: A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: ($\neg$ p V $\neg$ q V k). It has only one positive literal k.

It is equivalent to $p \wedge q \rightarrow k$.

What is Forward Chaining?

Forward chaining is a **data-driven** reasoning strategy in AI. It starts with known facts and applies rules to generate new facts or reach a conclusion. The process continues until no more new facts can be inferred or a goal is achieved. This approach is often used in expert systems for tasks such as troubleshooting and diagnostics.



Properties of Forward Chaining:

- **Data-Driven:** The reasoning starts from available data (facts) and works toward a goal.
- **Bottom-Up Approach:** It builds knowledge from facts, gradually moving towards conclusions.

In-Lab Discussion:

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.
```

Post-Lab Discussion:

Advantages of Forward Chaining

1. **Simplicity**: Forward chaining is straightforward and easy to implement.
2. **Automatic Data Processing**: It processes data as it arrives, making it suitable for dynamic environments where new data continuously becomes available.

3. **Comprehensive**: It explores all possible inferences, ensuring that all relevant conclusions are reached.
4. **Efficiency in Certain Scenarios**: It can be efficient when all possible inferences need to be made from a set of data.

Disadvantages of Forward Chaining

1. **Inefficiency in Goal-Oriented Tasks**: It can be inefficient if only a specific goal needs to be achieved, as it may generate many irrelevant inferences.
2. **Memory Intensive**: It can consume significant memory, storing a large number of intermediate facts.
3. **Complexity with Large Rule Sets**: As the number of rules increases, the system may become slow due to the need to check many conditions.

Difference Between Forwarding Chaining and Backward Chaining

Aspect	Forward Chaining	Backward Chaining
Approach	Data-Driven: Starts from facts and applies rules to reach conclusions.	Goal-Driven: Starts with a goal and works backward to verify if facts support it.
Search Strategy	Breadth-First Search: Explores multiple rules at the same level.	Depth-First Search: Focuses deeply on one path before trying others.
Direction	Bottom-Up: Moves from facts to conclusions.	Top-Down: Begins with the goal and works towards the facts.
Efficiency	May explore irrelevant rules, potentially reducing efficiency.	Risk of getting stuck in infinite loops if not managed properly.
Memory Usage	Can require more memory as it processes multiple rules at once.	More memory-efficient as it focuses on specific goals or sub-goals.
Complexity	Easier to implement for systems with many rules and data.	Can be more complex due to recursive searches for supporting facts.
Performance	Performs better when all relevant data is known upfront.	Works well when only specific information or goals are of interest.
Examples of Use Cases	Troubleshooting, diagnostics, and prediction systems.	Query systems, expert systems, and decision-making models.

Case-Based Discussion:

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."

Prove that **"Robert is criminal."** **Write a python program for the above statement using forward chaining.**

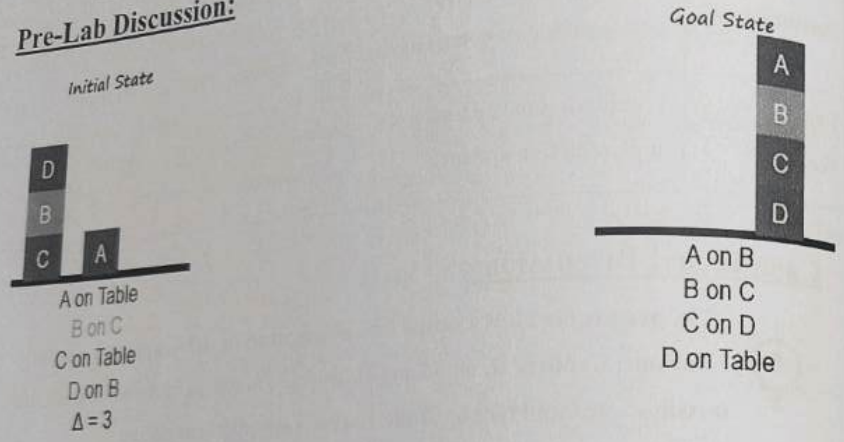
Roll No. :-

Implementation of Forward Chaining is executed Successfully and Verified.

Ex No: 3a

IMPLEMENTATION OF BLOCKS WORLD PROGRAM

Pre-Lab Discussion:



What is Blocks World Problem?

This is how the problem goes — There is a table on which some blocks are placed. Some blocks may or may not be stacked on other blocks. We have a robot arm to pick up or put down the blocks. The robot arm can move only one block at a time, and no other block should be stacked on top of the block which is to be moved by the robot arm.

Our aim is to change the configuration of the blocks from the Initial State to the Goal State, both of which have been specified in the diagram above.

If we want to know block word we must know planning in AI.

Planning

Planning refers to the process of computing several steps of a problem solving before executing any of them.Planning is useful as a problem solving technique for non decomposable problem.

Components of Planning System:

In any general problem solving systems, elementary techniques to perform following functions are required

- Choose the best rule (based on heuristics) to be applied
- Apply the chosen rule to get new problem state
- Detect when a solution has been found
- Detect dead ends so that new directions are explored.

To choose the rules ,

- first isolate a set of differences between the desired goal state and current state.
- identify those rules that are relevant to reducing these difference,
- if more rules are found then apply heuristic information to choose out of them.

To apply rules,

In simple problem solving system,

- applying rules was easy as each rule specifies the problem state that would result from its application.
- In complex problem we deal with rules that specify only a small part of the complete problem state.

Let us consider the famous problem name as **Block World Problem** ,which helps to understand the importance of planning in artificial intelligent system.

The block world environment has ,

- Square blocks of same size
- Blocks can be stacked one upon another.
- Flat surface (table) on which blocks can be placed.
- Robot arm that can manipulate the blocks. It can hold only one block at a time.

In block world problem, the state is described by a set of predicates representing the facts that were true in that state.One must describe for every action, each of the changes it makes to the state description.In addition, some statements that everything else remains unchanged is also [illegible] types of operations done by robot in block world

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):
```

```
def mov
  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'}
```

Post-Lab Discussion:

STRIPS

STRIPS stands for "STanford Research Institute Problem Solver," was the planner used in Shakey, one of the first robots built using AI technology ,which is an action-centric representation ,for each action , specifies the effect of an action.

A STRIPS planning problem specifies:
1) an initial state *S*
2) a goal *G*
3) a set of STRIPS actions

The **STRIPS representation** for an action consists of three lists,

- Pre_Cond list contains predicates which have to be true before operation.
- ADD list contains those predicates which will be true after operation
- DELETE list contain those predicates which are no longer true after operation

Predicates not included on either of these lists are assumed to be unaffected by the operation.Frame axioms are specified implicitly in STRIPS which greatly reduces amount of information stored.Let us discuss about the action lists for operations of block world problem.

Stack (X, Y)

Pre: CL (Y) ,HOLD (X)

Del: CL (Y), HOLD (X)

Add: AE , ON (X, Y)

UnStack (X, Y)

Pre: ON (X, Y) , CL (X) , AE

Del: ON (X, Y) , AE

Add: HOLD (X) , CL (Y)

Pickup (X)

Pre: ONT (X) , CL (X) ,AE

Del: ONT (X) , AE

Add: HOLD (X)

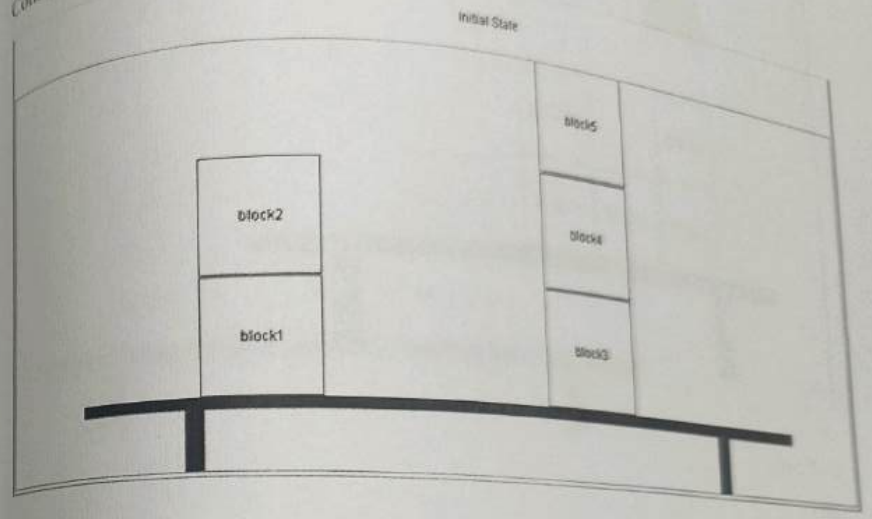
Putdown (X)

Pre: HOLD (X)

Del: HOLD (X)

Add: ONT (X) , AE

Consider a Block world problem ,



Initial State

on(block2, block1)

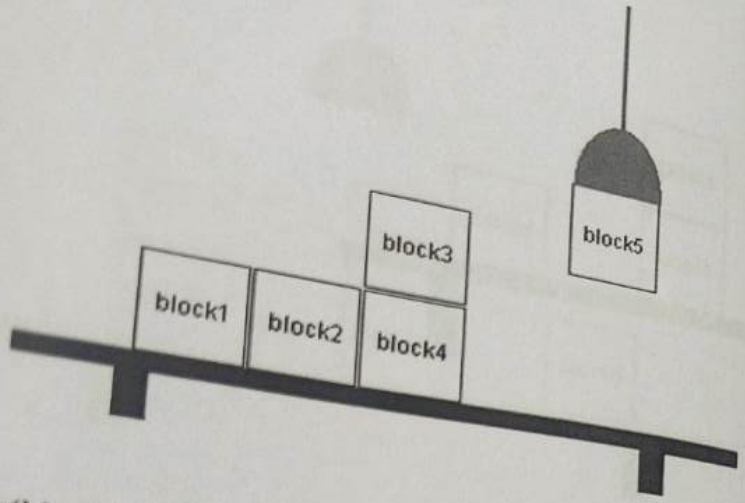
clear(block2)

ontable(block3)

on(block4, block3)

on(block5, block4)

clear(block5)

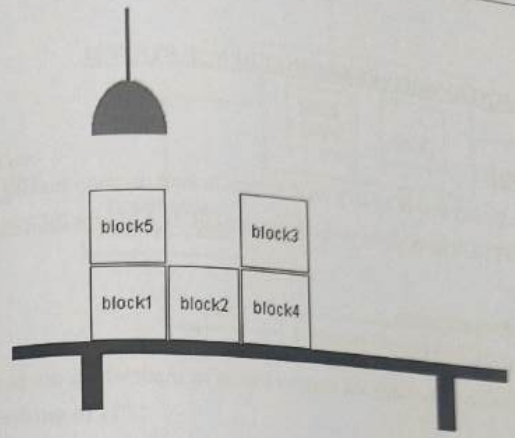


Stack(block5,block1)

Pre: CL (block1) ,HOLD (block5)

Del: CL (block1), HOLD (block5)

Add: AE , ON (block5, block1)



After completing all the operations what we found for the given problem,we had reaches the goal state.

armempty

on(block3, block4)

on(block5, block1)

ont(block2)

Case- Based Discussion:

Write a python program for STRIPS. Everything is given. Try to write a python code for it.

Result:- Implementation of Blocks world Program is executed Successfully and verified

Ex No: 3b

IMPLEMENTION OF A FUZZY INFERENCE SYSTEM

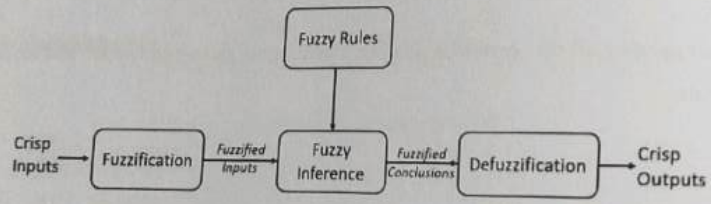
Pre-Lab Discussion:

Fuzzy Inference System is the key unit of a fuzzy logic system having decision making as its primary work. It uses the IFTHEN rules along with connectors OR or AND for drawing essential decision rules.

Characteristics of Fuzzy Inference System

Following are some characteristics of FIS –

- The output from FIS is always a fuzzy set irrespective of its input which can be fuzzy or crisp.
- It is necessary to have fuzzy output when it is used as a controller.
- A defuzzification unit would be there with FIS to convert fuzzy variables into crisp variables.

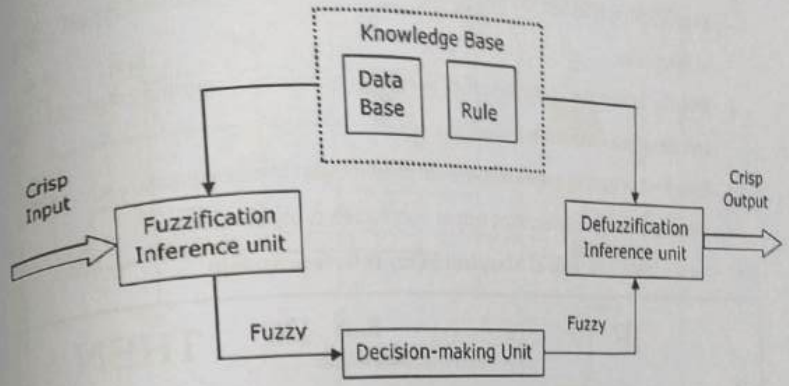


Functional Blocks of FIS

The following five functional blocks will help you understand the construction of FIS –

- **Rule Base** – It contains fuzzy IF-THEN rules.
- **Database** – It defines the membership functions of fuzzy sets used in fuzzy rules.
- **Decision-making Unit** – It performs operation on rules.
- **Fuzzification Interface Unit** – It converts the crisp quantities into fuzzy quantities.
- **Defuzzification Interface Unit** – It converts the fuzzy quantities into crisp quantities.

Following is a block diagram of fuzzy interference system.



Working of FIS

The working of the FIS consists of the following steps –

- A fuzzification unit supports the application of numerous fuzzification methods, and converts the crisp input into fuzzy input.
- A knowledge base - collection of rule base and database is formed upon the conversion of crisp input into fuzzy input.
- The defuzzification unit fuzzy input is finally converted into crisp output.

Methods of FIS

Let us now discuss the different methods of FIS. Following are the two important methods of FIS, having different consequent of fuzzy rules –

- Mamdani Fuzzy Inference System
- Takagi-Sugeno Fuzzy Model (TS Method)

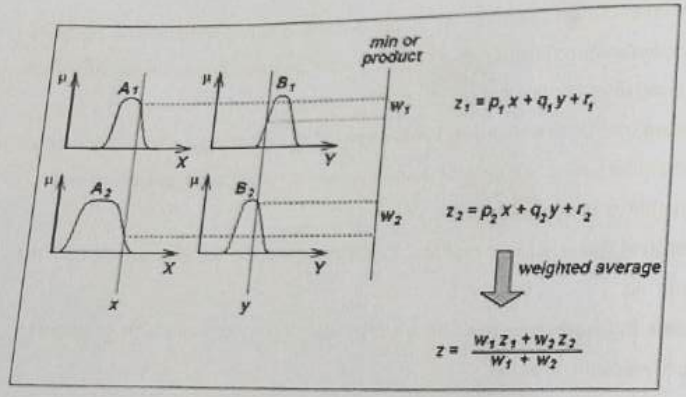
Mamdani Fuzzy Inference System

This system was proposed in 1975 by Ebhasim Mamdani. Basically, it was anticipated to control a steam engine and boiler combination by synthesizing a set of fuzzy rules obtained from people working on the system.

Steps for Computing the Output

Following steps need to be followed to compute the output from this FIS –

- **Step 1** – Set of fuzzy rules need to be determined in this step.
- **Step 2** – In this step, by using input membership function, the input would be made fuzzy.



Fuzzy reasoning procedure for a first-order Sugeno Fuzzy Model

How to Decide Whether to Apply- Mamdani or Sugeno Fuzzy Inference System?

- Mamdani technique is broadly acknowledged for capturing expert knowledge and information. It allows us to depict the skill in a more instinctive, more human-like way. However, Mamdani type fuzzy inference entails a considerable computational burden.
- On the other hand, the Sugeno method is computationally feasible. It functions effectively with advancement and versatile procedures making it exceptionally alluring in versatile issues, particularly for dynamic nonlinear frameworks.

In-Lab Discussion:

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
```

```
Predicted Performance Score: 67.85
```

Post-Lab Discussion:

Fuzzy Inference Systems Advantages

Fuzzy Inference System	Advantages
Mamdani	• Intuitive • Well-suited to human inputs • More interpretable and rule-based • Has widespread acceptance
Sugeno	• Computationally efficient • Functions well with linear techniques, like PID control • Functions with optimization and adaptive techniques • Guarantees output surface continuity • Well-suited to mathematical analysis

Comparison between the two methods

Let us now understand the comparison between the Mamdani System and the Sugeno Model.

- **Output Membership Function** – The main difference between them is on the basis of output membership function. The Sugeno output membership functions are either linear or constant.
- **Aggregation and Defuzzification Procedure** – The difference between them also lies in the consequence of fuzzy rules and due to the same their aggregation and defuzzification procedure also differs.
- **Mathematical Rules** – More mathematical rules exist for the Sugeno rule than the

Case-Based Discussion:

Try to write a python program for Driving problem. I have given the scenario now you have to choose any one method to solve the driving problem using python.

Example for fuzzy inference system:

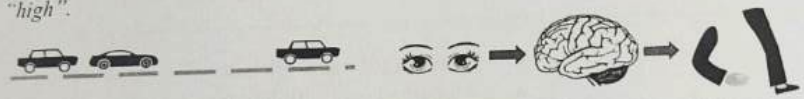
Driving problem: I am driving and want to keep a safety distance between cars. When the distance from the front car is *x*, what speed should I keep?

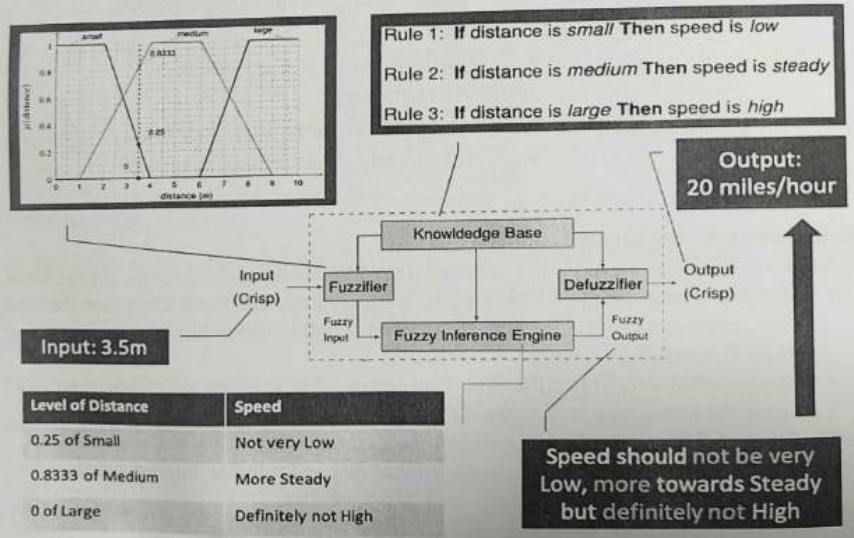
Linguistic Rules:

Rule 1: **If** distance is *small* **Then** speed is *low*
Rule 2: **If** distance is *medium* **Then** speed is *steady*
Rule 3: **If** distance is *large* **Then** speed is *high*

***More specific question**: When the distance from the front car is 3.5 m or so, what speed should I keep?*

***Answer**: The speed should be not very "low", more toward "steady" but definitely not "high".*





Level of Distance	Speed
0.25 of Small	Not very Low
0.8333 of Medium	More Steady
0 of Large	Definitely not High

Speed should not be very Low, more towards Steady but definitely not High

Result:

Implementation of Fuzzy Inference system is created Successfully and verified.

2. What is an intelligent

Answer: An intelligent agent is a system that perceives its environment and t 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, 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 rule 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 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 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 infer and working memory.