Ex No: 1

IMPLEMENT EIGHT QUEENS PROBLEM

Pre-Lab Discussion:

Backtracking algorithms are like problem-solving strategies that help explore different options to find the best solution. They work by trying out different paths and if one doesn't work, they backtrack and try another until they find the right one. It's like solving a puzzle testing different pieces until they fit together perfectly.

How Does a Backtracking Algorithm Work?

A **backtracking algorithm** works by recursively exploring all possible solutions to a problem. It starts by choosing an initial solution, and then it explores all possible extensions of that solution. If an extension leads to a solution, the algorithm returns that solution. If a extension does not lead to a solution, the algorithm backtracks to the previous solution and tries a different extension.

The following is a general outline of how a backtracking algorithm works:

1. Choose an initial solution.
2. Explore all possible extensions of the current solution.
3. If an extension leads to a solution, return that solution.
4. If an extension does not lead to a solution, backtrack to the previous solution and different extension.
5. Repeat steps 2-4 until all possible solutions have been explored.

When to Use a Backtracking Algorithm?

Backtracking algorithms are best used to solve problems that have the following characteristics:

- There are multiple possible solutions to the problem.
- The problem can be broken down into smaller subproblems.
- The subproblems can be solved independently.

```
# 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 limited resources (bandwidth, memory, etc.) without interference.
- **Logistics Optimization:** Planning routes and layouts to prevent conflicts and 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 avoiding obstacles.
- **Automated Reasoning Systems:** Solving logical puzzles and proving theorems by exploring non-conflicting states.
- **VLSI Design:** Placing components on integrated circuits without overlaps or electrical conflicts.
- **Software Engineering (Algorithm Design & Scalability):** Understanding and 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 after removing the symmetrical solutions due to rotation. For 8*8 chess board with 8 queens there 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 implementation of Eight queens problem Succesfully compiled.

Ex No: 1b

IMPLEMENTATION OF DEPTH FIRST SEARCH

Pre-Lab Discussion:

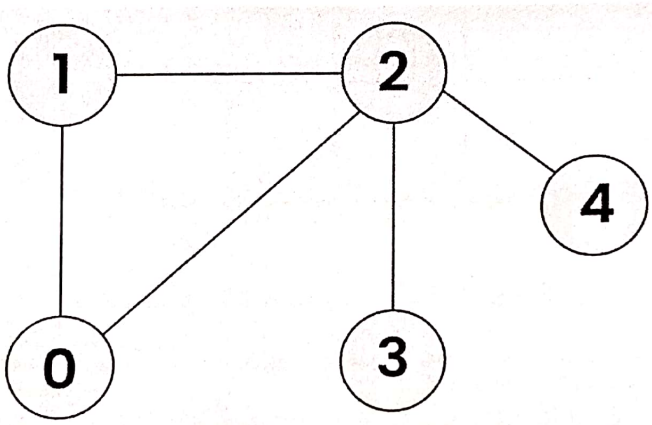
Depth First Search or DFS for a Graph – Python

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'
```

- **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:-

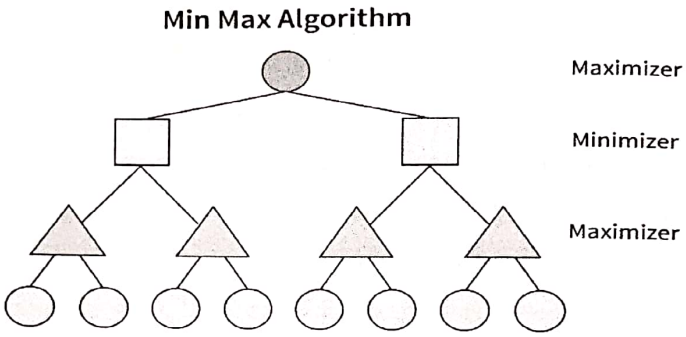
The implementation of depth first search was successfully compiled.

Ex No: 1c

IMPLEMENTATION OF MINIMAX algorithm

Pre-Lab Discussion:

The minimax algorithm in game theory helps the players in two-player games decide best move. It is crucial to assume that the other player is also making the best move w determining the best course of action for the present player. Each player attempts to r the maximum loss that they can suffer in the game. This article will cover the minima 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 aim 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 maximi assuming the opponent will play optimally.

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

3. Create `isMovesLeft(board)` to check for empty spaces; return `True` if moves a

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 va return `best`.
- If `isMax` is `False` (Human's turn), initialize `best = +∞`, loop through empt place `O`, call `minimax(board, True)`, undo move, update `best` with minim and return `best`.

5. Implement `findBestMove(board)`:

- Initialize `bestVal = -∞` and `bestMove = (-1, -1)`.
- Loop through empty cells, place `X`, call `minimax(board, False)`, undo m `bestMove` if a better move is found.
- Return `bestMove`.

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

7. Initialize a sample `board`, print its state, call `findBestMove(board)`, update the 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 i
    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

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:

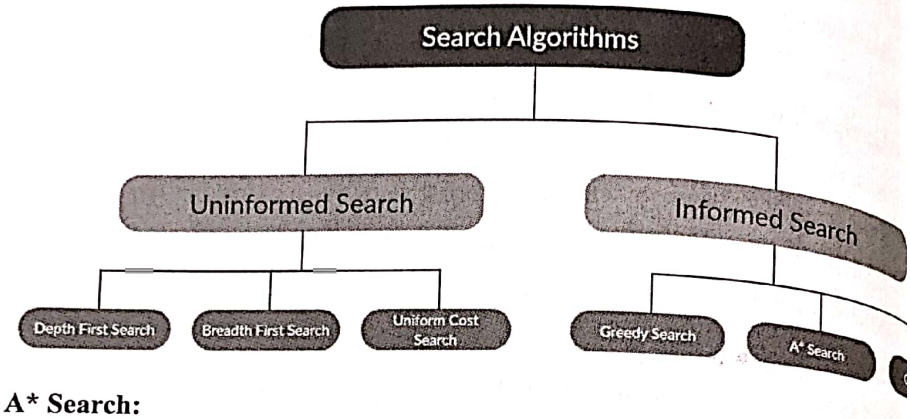
Write a python program for poker game using minmax algorithm.

Result:- The implementation of minmax algorithm was succesfully compiled.

Ex No: 1d

IMPLEMENTATION OF A* SEARCH ALGORITHM

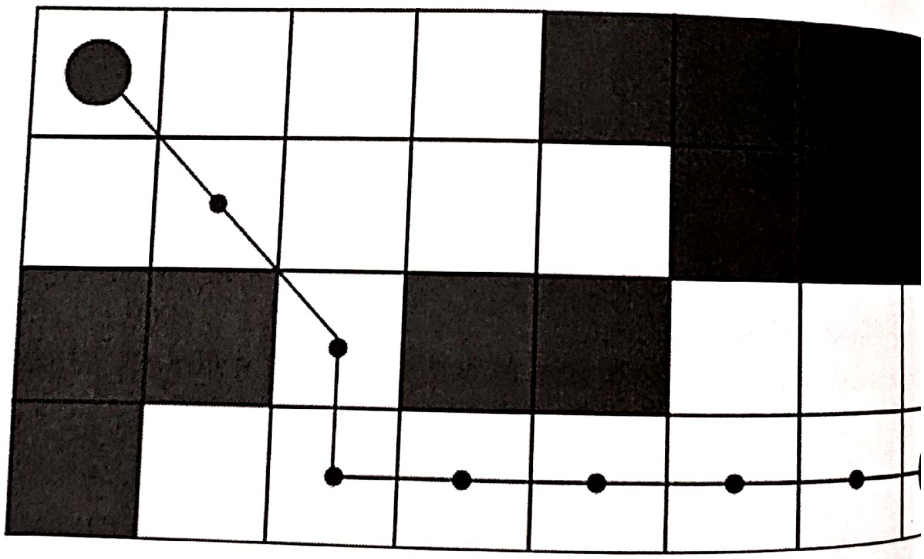
Pre-Lab Discussion:



A* Search:

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

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



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

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 was succesfully compiled.

Ex No: 2

IMPLEMENTATION OF DECISION MAKING AND KNOWLEDGE REPRESENTATION

Pre-Lab Discussion:

What Is Prolog?

Prolog is a declarative and logic programming language designed for developing logic-based AI applications. Developers can set rules and facts around a problem, and then Prolog's 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 is Paris." Rules are logical statements that describe the relationships between different facts, 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 example, a 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 uses the not_same/2 predicate to state that if X is a man and Y is a woman, then X is not the same as Y.

How Is Prolog Different From Other Programming Languages?

Prolog is a declarative programming language, while common programming languages like Python or JavaScript are imperative. A declarative language tells a computer what it should do, and focuses on the goal itself without providing steps to get to the goal. In contrast, an imperative language tells a computer how to reach a goal step-by-step.

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 roads and 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. As they 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 knowledge representation. One critical module in such systems is the **lane change decision making** component. This module determines whether it's legal and safe for the vehicle to change lanes based on current road conditions and traffic rules. **Write a Prolog program for that Module.**

Result:-

The implementation of decision making and knowledge representation was successfully compailed.

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, return `None` (unification fails).

2. **Define the variable unification function (`unify_var`):**
   - If the variable already exists in the substitution set, apply unification recur
   - 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** 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:
```

Case-Based Discussion:

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

Result:-

the implementation of unification and resolution algorithm was succesfully compiled.

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.

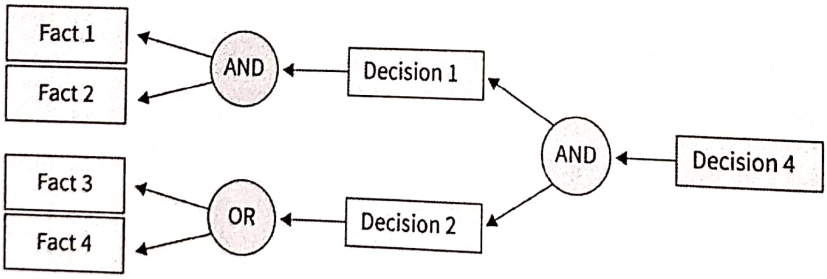
Horn clause: A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: (¬ p V ¬ q V 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.

- `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:-

The implementation of Backward Chaining was succesfully compiled.

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.

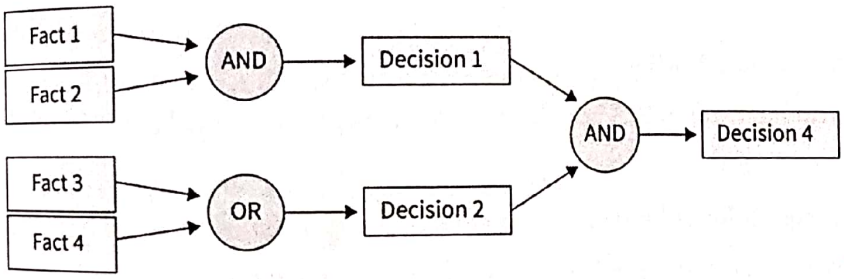
Horn clause: A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: (¬ p V ¬ 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.

```
# 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.

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.**

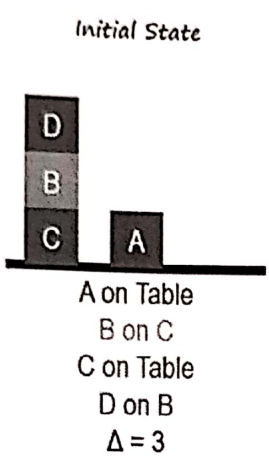
Result:-

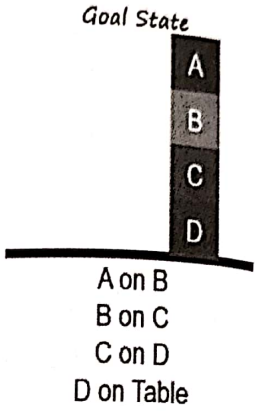
The implementation of Forward Chaining was Successfully Compailed.

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:

```
        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:

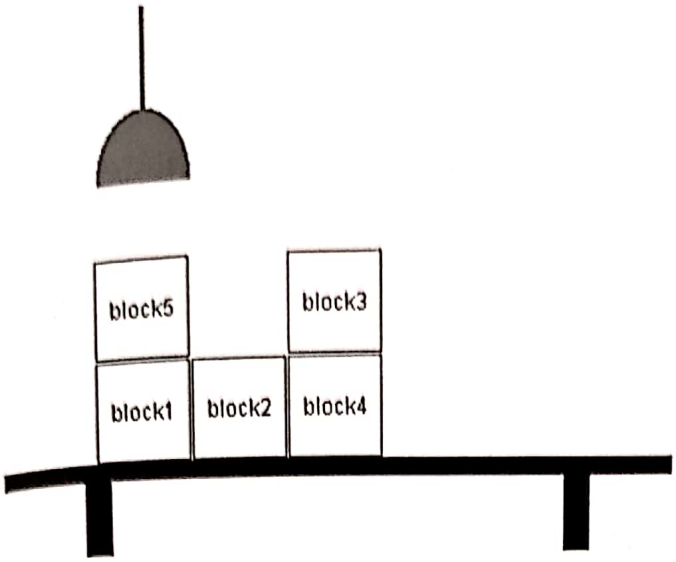
```
Initial State: {'A': 'B', 'B': 'table', 'C': 'table'}

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'}
```

Post-Lab Discussion:

STRIPS



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: The implementation of blocks world program was successfully compiled

Ex No: 3b

IMPLEMENTION OF A FUZZY INFERENCE SYSTEM

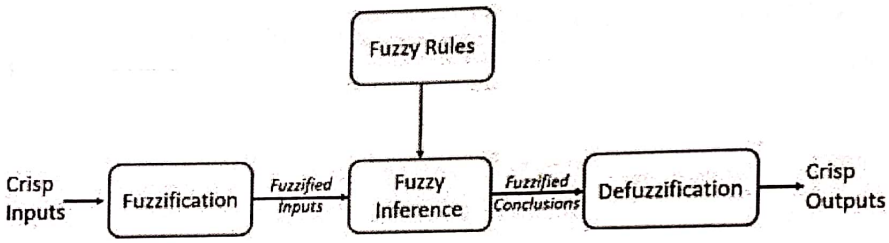
Pre-Lab Discussion:

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.

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')
```

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

Output

```
Predicted Performance Score: 67.85
```

Result:- The implementation of a Fuzzy Inference system was succesfully compailed.

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 Mamdani rule.