SREE BUDDHA COLLEGE OF ENGINEERING

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

AIL 333 AI ALGORITHMS LAB

LAB MANUAL

Prepared By	Approved By

LIST OF EXPERIMENTS

1.Installation and working on various AI tools.

2. Implement basic search strategies for selected AI applications

Uninformed Searches

- Breadth First Search
- Depth First Search

3. Implement state space search algorithms

    1. Vacuum Cleaner World
    2. 8 Queens Problem

4. Implement informed search algorithms

    1. A* Algorithm
    2. Best First Search Algorithm

5. Implement backtracking algorithms for CSP

6. Implement local search algorithms for CSP

7. Implement propositional logic inferences for AI tasks

8. Implementation of Knowledge representation schemes

9. Implement travelling salesman problem

10. Implementation of Game playing (adversarial search)

    1. MIN MAX Game Playing with Alpha-Beta Pruning

11. Mini Project

AIL 333	AI ALGORITHMS LAB	CATEGORY	L	T	P	Credit	Year of Introduction
		PCC	0	0	3	2	2022

Preamble:

This laboratory course enables the students to get the fundamental concepts in the area of Artificial Intelligence. This course covers the AI based Algorithms, logical reasoning agents and implementation of these reasoning systems using either backward or forward inference mechanisms. This course helps the learners to apply AI techniques to solve real world problems.

Prerequisite: A sound knowledge of the basics of programming, Discrete Mathematics.

Course Outcomes: After the completion of the course, the student will be able to:

CO#	Course Outcomes
CO1	State the basics of learning problems with hypothesis and version spaces **(Cognitive Knowledge Level: Understand).**
CO2	Demonstrate real-world problems as state space problems, optimization problems or constraint satisfaction problems. **(Cognitive Knowledge Level: Apply)**
CO3	Simulate given problem scenario and analyze its performance. **(Cognitive Knowledge Level: Apply)**
CO4	Develop programming solutions for given problem scenario. **(Cognitive Knowledge Level: Apply)**
CO5	Design and develop an expert system by using appropriate tools and techniques. **(Cognitive Knowledge Level: Apply)**

Mapping of course outcomes with program outcomes

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	✓	✓	✓					✓		✓		✓
CO2	✓	✓	✓	✓				✓		✓		✓
CO3	✓	✓	✓	✓	✓			✓		✓		✓
CO4	✓	✓	✓	✓	✓	✓		✓		✓		✓
CO5	✓	✓	✓	✓	✓			✓		✓		✓

Abstract POs defined by National Board of Accreditation			
PO#	**Broad PO**	**PO#**	**Broad PO**
PO1	Engineering Knowledge	**PO7**	Environment and Sustainability
PO2	Problem Analysis	**PO8**	Ethics
PO3	Design/Development of solutions	**PO9**	Individual and teamwork
PO4	Conduct investigations of complex problems	**PO10**	Communication
PO5	Modern tool usage	**PO11**	Project Management and Finance
PO6	The Engineer and Society	**PO12**	Lifelong learning

Assessment Pattern

Bloom's Category	**Continuous Assessment Test (Internal Exam) Marks in percentage**	**End Semester Examination Marks in percentage**
Remember	**20**	**20**
Understand	**20**	**20**
Apply	**60**	**60**
Analyze		
Evaluate		
Create		

Mark Distribution

Total Marks	**CIE Marks**	**ESE Marks**	**ESE Duration**
150	**75**	**75**	**3 hours**

Continuous Internal Evaluation Pattern:

Attendance	: **15 marks**
Continuous Evaluation in Lab	: **30 marks**
Continuous Assessment Test	: **15 marks**
Viva voce	: **15 marks**

Internal Examination Pattern:

The internal examination shall be conducted for 100 marks, which will be converted to out of 15, while calculating internal evaluation marks. The marks will be distributed as, Algorithm - 30 marks, Program - 20 marks, Output - 20 marks and Viva - 30 marks.

End Semester Examination Pattern:

The end semester examination will be conducted for a total of 75 marks and shall be distributed as, Algorithm - 30 marks, Program - 20 marks, Output - 20 marks and Viva- 30 marks.

Operating System to Use in Lab	: Linux/Windows
Programming Language to Use in Lab	: C++/Java/Python/Prolog

Fair Lab Record:

All the students attending the Artificial Intelligence Algorithms laboratory should have a fair record. Every experiment conducted in the lab should be noted in the fair record. For every experiment, in the fair record, the right-hand page should contain experiment heading, experiment number, date of experiment, aim of the experiment, procedure/algorithm followed, other such details of the experiment and final result. The left-hand page should contain a print out of the respective code with sample input and corresponding output obtained. All the experiments noted in the fair record should be verified by the faculty regularly. The fair record, properly certified by the faculty, should be produced during the time of end semester examination for the verification by the examiners.

Syllabus

***Mandatory**

12. Installation and working on various AI tools viz. Python, R, GATE, NLTK, MATLAB etc.*****
13. Implement basic search strategies for selected AI applications*****.
14. Implement state space search algorithms*****
15. Implement informed search algorithms*****
16. Implement backtracking algorithms for CSP*****
17. Implement local search algorithms for CSP*****
18. Implement propositional logic inferences for AI tasks*****
19. Implementation of Knowledge representation schemes*****
20. Implement travelling salesman problem*****
21. Implementation of Game playing (adversarial search)
22. Mini Project

References:

1. Dan W. Patterson, “Introduction to AI and ES”, Pearson Education, 2007
2. Kevin Night, Elaine Rich, and Nair B., “Artificial Intelligence”, McGraw Hill, 2008
3. Patrick H. Winston, "Artificial Intelligence", Third edition, Pearson Edition, 2006
4. Deepak Khemani, “Artificial Intelligence”, Tata McGraw Hill Education, 2013 (http://nptel.ac.in/)
5. Artificial Intelligence by Example: Develop machine intelligence from scratch using real artificial intelligence use cases -by Dennis Rothman, 2018
6. Padhy, N.P. 2009. Artificial Intelligence and Intelligent Systems, Oxford University Press
7. Brachman, R. and Levesque, H. 2004. Knowledge Representation and Reasoning, Morgan Kaufmann.

PRACTICE QUESTIONS

1. Implementation of Depth-First Search (DFS).
2. Write a program to implement water jug problem.
3. Implement variants of hill-climbing and genetic algorithms.
4. Implement tic tac toe game for 0 and X.
5. Develop a program to construct a pruned game tree using Alpha-Beta pruning. Take the sequence, [5, 3, 2, 4, 1, 3, 6, 2, 8, 7, 5, 1, 3, 4] of MINIMAX values for the nodes at the cutoff depth of 4 plies. Assume that branching factor is 2, MIN makes the first move, and nodes are generated from right to left.
6. Write a program to implement production system.
7. Write a program to implement heuristic search procedure.
8. Write a program to implement Expert system.
9. Write a program to implement search problem of 3 x 3 puzzles.

EXP NO 1: INTRODUCTION TO ARTIFICIAL INTELLIGENCE TOOLS

ARTIFICIAL INTELLIGENCE

Artificial intelligence, or AI, is a simulation of intelligent human behavior. It’s a computer or system designed to perceive its environment, understand its behaviors, and take action. Consider self-driving cars: AI-driven systems like these integrate AI algorithms, such as machine learning and deep learning, into complex environments that enable automation.

ARTIFICIAL INTELLIGENCE TOOLS

Artificial Intelligence has facilitated the processing of a large amount of data and its use in the industry. The number of tools and frameworks available to data scientists and developers has increased with the growth of AI and ML. This article on Artificial Intelligence Tools & Frameworks will list out some of these in the following sequence:

LIST OF AI TOOLS

Scikit Learn

Scikit-learn is one of the most well-known ML libraries. It underpins many administered and unsupervised learning calculations. Precedents incorporate direct and calculated relapses, choice trees, bunching, k-implies, etc. It expands on two essential libraries of Python, NumPy and SciPy. It includes a lot of calculations for regular AI and data mining assignments, including bunching, relapse and order. Indeed, even undertakings like changing information, feature determination and ensemble techniques can be executed in a couple of lines. For a fledgeling in ML, Scikit-learn is a more-than-adequate instrument to work with, until you begin actualizing progressively complex calculations.

Tensorflow

It utilizes an arrangement of multi-layered hubs that enables you to rapidly set up, train, and send counterfeit neural systems with huge datasets. This is the thing that enables Google to recognize questions in photographs or comprehend verbally expressed words in its voiceacknowledgment application.

PyTorch

PyTorch is an AI system created by Facebook. Its code is accessible on GitHub and at the present time has more than 22k stars. It has been picking up a great deal of energy since 2017 and is in a relentless reception development.

OpenNN

Jumping from something that is completely beginner friendly to something meant for experienced developers, OpenNN offers an arsenal of advanced analytics. It features a tool, Neural Designer for advanced analytics which provides graphs and tables to interpret data entries.

H20: Open Source AI Platform

H20 is an open-source deep learning platform. It is an artificial intelligence tool which is business oriented and help them to make a decision from data and enables the user to draw insights. There are two open source versions of it: one is standard H2O and other is paid version Sparkling Water. It can be used for predictive modelling, risk and fraud analysis, insurance analytics, advertising technology, healthcare and customer intelligence.

Google ML Kit

Google ML Kit, Google’s machine learning beta SDK for mobile developers, is designed to enable developers to build personalised features on Android and IOS phones. The kit allows developers to embed machine learning technologies with app-based APIs running on the device or in the cloud. These include features such as face and text recognition, barcode scanning, image labelling and more. Developers are also able to build their own TensorFlow Lite models in cases where the builtin APIs may not suit the use case.

GATE

GATE (General Architecture for Text Engineering) is a powerful and widely used framework in Artificial Intelligence (AI) for processing and analyzing text. It provides a comprehensive suite of tools and resources for a variety of Natural Language Processing (NLP) tasks, making it a valuable asset in AI research and applications.

RESULT

Various AI tools are familiarized.

EXP NO:2 STATE SPACE SEARCH ALGORITHMS

2.1 VACCUM CLEARNER WORLD

Algorithm

1. Start
2. initializing goal_state
3. 0 indicates Clean and 1 indicates Dirty
4. Goal state is defined as goal_state = {'A': '0', 'B': '0'} and cost = 0
5. user_input of location vacuum is placed
6. user_input if location is dirty or clean
7. if location_input is 'A',Location ,A is Dirty.
8. suck the dirt and mark it as clean
9. calculate cost.
10. goal_state['A'] = '0'
11. if location_input is 'A',Location ,A is Clean
12. Do Nothing.
13. Moving right to the Location B.
14. cost for moving right
15. If dirty suck the dirt and mark it as clean
16. goal_state['B'] = '0'
17. cost for suck
18. If B is clean Do Nothing
19. Else if location_input is 'B',Location ,B is Dirty.
20. suck the dirt and mark it as clean
21. calculate cost.
22. goal_state['B'] = '0'
23. if location_input is 'B',Location ,B is Clean
24. Do Nothing.
25. Moving Left to the Location A.
26. cost for moving left
27. If dirty suck the dirt and mark it as clean
28. goal_state['A'] = '0'
29. cost for suck
30. If A is clean Do Nothing
31. Print Goal State
32. Print Final Cost as Performance Measurement
33. Stop

PROGRAM

```
def vacuum_world():
        # initializing goal_state
        # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum") #user_input of location vacuum is placed
    status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean
    status_input_complement = input("Enter status of other room")
    print("Initial Location Condition" + str(goal_state))

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['A'] = '0'
            cost += 1                      #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                # if B is Dirty
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1                      #cost for moving right
                print("COST for moving RIGHT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1                   #cost for suck
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action" + str(cost))
                # suck and mark clean
                print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':# if B is Dirty
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1                   #cost for moving right
                print("COST for moving RIGHT " + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1                   #cost for suck
                print("Cost for SUCK" + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action " + str(cost))
                print(cost)
                # suck and mark clean
                print("Location B is already clean.")

    else:
        print("Vacuum is placed in location B")
        # Location B is Dirty.
        if status_input == '1':
            print("Location B is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['B'] = '0'
            cost += 1  # cost for suck
            print("COST for CLEANING " + str(cost))
            print("Location B has been Cleaned.")

            if status_input_complement == '1':
                # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1  # cost for moving right
                print("COST for moving LEFT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("COST for SUCK " + str(cost))
                print("Location A has been Cleaned.")

        else:
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

            if status_input_complement == '1':  # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1  # cost for moving right
                print("COST for moving LEFT " + str(cost))
                # suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("Cost for SUCK " + str(cost))
                print("Location A has been Cleaned. ")
            else:
                print("No action " + str(cost))
                # suck and mark clean
                print("Location A is already clean.")

        # done cleaning
        print("GOAL STATE: ")
        print(goal_state)
        print("Performance Measurement: " + str(cost))



    vacuum_world()
```

RESULT

The above program has been successfully executed and output obtained is verified.

OUTPUT

```
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
```

```
Location B has been Cleaned.

GOAL STATE:

{'A': '0', 'B': '0'}

Performance Measurement: 3
```

EXP 2: STATE SPACE SEARCH ALGORITHMS

2. 2 N QUEENS PROBLEM

```
1. Start
2.  Taking number of queens as input from user and create a chessboard. The board is
initialized with zeros, representing empty cells.
        NxN matrix with all elements set to 0
3. Checking vertically and horizontally, this function checks if placing a queen at (i, j) would
be attacked by any existing queens. It checks rows, columns, and diagonals.
4. Define k ranging from 0to N
        check if board[i][k]==1 or board[k][j]==1:
        Then return True
5. checking diagonally
        Define k ranging from 0to N
        Define I ranging from 0to N
        Check if  (k+l==i+j) or (k-l==i-j):
        Check if board[k][l]==1:
        Then return True
        Else return False
6. Placing Queens on board, the recursive function tries to place queens one by one in
different cells, ensuring no two queens attack each other. It uses backtracking to find a
solution.
        Define I ranging from 0to N
        Define J ranging from 0to N
        Check if (not(attack(i,j))) and (board[i][j]!=1):
        board[i][j] = 1
        if N_queens(n-1)==True:
        Then return True
        board[i][j] = 0
        Then return False
7. Display Queens on Board
8. Stop
```

PROGRAM

```
# Taking number of queens as input from user

print ("Enter the number of queens")

N = int(input())

# here we create a chessboard

# NxN matrix with all elements set to 0

board = [[0]*N for _ in range(N)]
```

```
def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queens(N)
for i in board:
    print (i)
```

RESULT

The above program has been successfully executed and output obtained is verified.

OUTPUT

```
Enter the number of queens
8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

EXP NO: 3 UNINFORMED SEARCHES

3.1. Breadth First Search Algorithm

1. **START**
2. **Enqueue the Start State**:
   - Enqueue the start state into the queue.
   - Mark the start state as visited by adding it to the visited set.
   - Initialize the predecessor of the start state as `None`
3. **While the Queue is Not Empty**:
   - Dequeue a node(current_node) from the front of the queue.
   - If `current_node` is the goal state, break the loop
4. **For Each Neighbor of the `current_node`**:
   - If the neighbor has not been visited:
     - Enqueue the neighbor.
     - Mark the neighbor as visited.
     - Set the predecessor of the neighbor as the `current_node`.
5. If the goal state is reached, reconstruct the path from the start state to the goal state using the predecessor dictionary.
   - **Reconstruction of path**: Initialize an empty list to store the path.
     - If the goal node was visited:
     - Starting from the goal node, trace back to the start node using the predecessor dictionary.
     - Append each node to the path list.
     - Reverse the path list to get the correct order from start to goal.
6. Return the path from the start state to the goal state if found, otherwise return a message indicating no path exists.
7. **STOP**

```
from collections import deque


def bfs(graph, start, goal):

    # Initialize the queue, visited set, and predecessor dictionary

    queue = deque([start])

    visited = set([start])

    predecessor = {start: None}


    # While the queue is not empty

    while queue:

        current_node = queue.popleft()
        # If the goal state is found, break the loop
        if current_node == goal:
            break

        # For each neighbor of the current node
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
                predecessor[neighbor] = current_node

    # Path reconstruction
    path = []
    if goal in visited:
        current = goal
        while current is not None:
            path.append(current)
            current = predecessor[current]
        path.reverse()
        return path
    else:
        return "No path exists"

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
    'G': []
}


start = 'A'
goal = 'F'
path = bfs(graph, start, goal)
print("Path from start to goal:", path)
```

OUTPUT

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

DFS

Step 1: Initialize the Open and Closed Lists

- **Open List**: This is a stack data structure used to keep track of nodes that need to be explored. We start by adding the `start_node` to this list.
- **Closed List**: This is a set used to keep track of nodes that have already been visited to avoid re-processing them.
- Path Dictionary: Maps each node to its predecessor to reconstruct the path later

Step 2: Loop Until Open List is Empty

- The main loop runs as long as there are nodes in the open list.

Step 3: Pop a Node from the Open List

- The last node (most recently added) is removed from the open list. This ensures that we are exploring as deep as possible along each branch before backtracking.

Step 4: Check if the Current Node is the Goal Node

- If the current node is the goal node, we can stop the search and return a message or the path found.

Step 5: Process the Current Node if Not Visited

- If the current node is not in the closed list (i.e., not visited):
    - Add it to the closed list.
    - Retrieve its adjacent nodes (children).

Step 6: Add Children to the Open List

- For each child of the current node:
    - If the child has not been visited (not in the closed list), add it to the open list.
    - This ensures that unvisited children will be explored in subsequent iterations.

Step 7: Repeat Until Open List is Empty or Goal is Found

- Continue the loop until there are no more nodes to explore or the goal node is found.

Step 8: **Reconstruct Path Function**:

- If the goal node is found, backtrack from the `goal_node` to the `start_node` using the path dictionary.

- Reverse the path to get it in the correct order from `start_node` to `goal_node`.

Step 9: Return Result:

- Return the path if found, otherwise indicate that no path exists.

```
def depth_first_search(graph, start_node, goal_node):
    # Open list: stack to keep track of nodes to explore
    open_list = [start_node]
    # Closed list: set to keep track of visited nodes
    closed_list = set()
    # Dictionary to store the path
    path = {start_node: None}

    while open_list:
        # Pop the last node from the stack
        current_node = open_list.pop()

        # If the goal node is found, reconstruct the path
        if current_node == goal_node:
            return reconstruct_path(path, start_node, goal_node)

        # If the current node has not been visited
        if current_node not in closed_list:
            # Add the current node to the closed list
            closed_list.add(current_node)

            # Get all adjacent nodes (children) of the current node
            children = graph.get(current_node, [])

            # Add each child to the open list (stack) if not visited
            for child in children:
                if child not in closed_list:
                    open_list.append(child)
                    # Store the path
                    path[child] = current_node
    return f"Goal node {goal_node} not found in the graph."

def reconstruct_path(path, start_node, goal_node):
    # Reconstruct the path from goal_node to start_node
    current_node = goal_node
    reversed_path = []
    while current_node is not None:
        reversed_path.append(current_node)
        current_node = path[current_node]
    # Return the reversed path (from start_node to goal_node)
    return list(reversed(reversed_path))

# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
    'G': [],
}

# Perform DFS
start_node = 'A'
goal_node = 'F'
result = depth_first_search(graph, start_node, goal_node)
print("Path from start to goal: ",result)  # Output: ['A', 'C', 'F']
```

OUTPUT

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

BEST FIRST SEARCH

1. START
2. Insert the `start` node into the open list with its heuristic value as the priority.
3. **While Open List is Not Empty**:
    a. **Expand Node**:
        i. Remove the node with the lowest heuristic value from the open list. This is the node that is currently considered the most promising to reach the goal.
        ii. Print or record the node being expanded for debugging or information purposes.
    b. **Check for Goal**:
        i. If the current node is the `goal` node, print that the goal has been found and terminate the search.
    c. **Mark Node as Explored**:
        i. Add the current node to the closed list to mark it as explored.
    d. **Process Neighbors**:
        i. For each neighbor of the current node:
            1. **If Neighbor is in Closed List**: Skip this neighbor since it has already been explored.
            2. **If Neighbor is Not in Open List**:
                a. Add the neighbor to the open list with its heuristic value as the priority.
            3. **If Neighbor is Already in Open List**: This step is usually implicit since the neighbor is added only if not already present.
4. If the open list is empty and the goal has not been found, print that the goal node is not reachable or not found.
5. STOP

```
class Node:
    def __init__(self, name, heuristic):
        self.name = name
        self.heuristic = heuristic
        self.neighbors = {}

    def add_neighbor(self, neighbor, cost):
        self.neighbors[neighbor] = cost

def best_first_search(start, goal):
    open_list = [start]  # List of nodes to be explored
    closed_list = set()  # Set of nodes that have been explored

    while open_list:
        # Choose the node with the smallest heuristic value
        current_node = min(open_list, key=lambda node: node.heuristic)
        open_list.remove(current_node)

        print(f"Expanding node: {current_node.name}")
        # Check if the current node is the goal
        if current_node == goal:
            print(f"Goal node {goal.name} found!")
            return

        closed_list.add(current_node)

        # Add neighbors to the open list
        for neighbor in current_node.neighbors:
            if neighbor in closed_list or neighbor in open_list:
                continue
            open_list.append(neighbor)

    print("Goal node not found.")

# Example Usage
if __name__ == "__main__":
    # Creating nodes with heuristic values
    a = Node('A', 5)
    b = Node('B', 3)
    c = Node('C', 1)
    d = Node('D', 2)
    e = Node('E', 0)

    # Adding neighbors (node, cost)
    a.add_neighbor(b, 1)
    a.add_neighbor(c, 4)
    b.add_neighbor(d, 2)
    c.add_neighbor(e, 3)
    d.add_neighbor(e, 1)

    # Performing Best First Search
    best_first_search(a, e)
```

OUTPUT

```
Expanding node: A
Expanding node: C
Expanding node: E
Goal node E found!
```

1. START
2. **While Open List is Not Empty**:

   Pop Node with Minimum `f` Value:

   - Remove the node from the open list that has the smallest `f` value (`g + heuristic`). This node is considered the most promising to expand next.
   - Print the name of the node being expanded for tracking purposes.

   Check for Goal Node:

   - If the current node is the `goal` node:
     - Print a message indicating that the goal has been found.
     - Call the `reconstruct_path` function to retrieve the path from the start to the goal and return it.

   Mark Node as Explored:

   - Add the current node to the closed list to mark it as fully explored.

   Expand Neighbors:

   - For each neighbor of the current node:
     - **Skip if Neighbor is in Closed List**:
       - If the neighbor has already been explored (i.e., is in the closed list), skip it.
     - **Calculate Tentative `g` Value**:
       - Calculate the tentative cost (`tentative_g`) to reach the neighbor node as the sum of the current node's cost (`g`) and the cost to reach this neighbor.
     - **Update Neighbor's Cost and Parent**:
       - If the neighbor is not in the open list or the newly calculated cost (`tentative_g`) is lower than the neighbor's current cost (`g`):
         - Update the neighbor's cost (`g`) to the `tentative_g`.
         - Set the parent of the neighbor to the current node.
         - If the neighbor is not already in the open list, add it to the open list.
3. **If Open List is Empty**:If the open list becomes empty and the goal has not been reached: Print a message indicating that the goal node was not found.
4. **Reconstruct Path**:

   Trace Back from Goal Node:Start from the `goal` node and trace back to the start node using the `parent` references.

Build Path:Collect the names of the nodes in a list as you trace back.Reverse the list to get the path from the start node to the goal node.

Return Path:

Return the reconstructed path.

5. STOP

```
import heapq

class Node:
    def __init__(self, name, heuristic):
        self.name = name
        self.heuristic = heuristic
        self.neighbors = {}  # Dictionary of {neighbor_node: cost}
        self.g = float('inf')  # Cost from start to this node
        self.parent = None

    def add_neighbor(self, neighbor, cost):
        self.neighbors[neighbor] = cost

    def __lt__(self, other):
        # Compare nodes based on f = g + heuristic
        return (self.g + self.heuristic) < (other.g + other.heuristic)

def a_star_search(start, goal):
    open_list = []
    closed_list = set()

    start.g = 0
    heapq.heappush(open_list, start)

    while open_list:
        current_node = heapq.heappop(open_list)
        print(f"Expanding node: {current_node.name}")

        if current_node == goal:
            return reconstruct_path(goal)

        closed_list.add(current_node)

        for neighbor, cost in current_node.neighbors.items():
            if neighbor in closed_list:
                continue

            tentative_g = current_node.g + cost

            if tentative_g < neighbor.g:
                neighbor.g = tentative_g
                neighbor.parent = current_node
                if neighbor not in open_list:
                    heapq.heappush(open_list, neighbor)
    print("Goal node not found.")
    return None

def reconstruct_path(goal):
    path = []
    current = goal
    while current:
        path.append(current.name)
        current = current.parent
    path.reverse()
    return path

# Example Usage
if __name__ == "__main__":
    # Create nodes with heuristic values
    a = Node('A', 5)
    b = Node('B', 3)
    c = Node('C', 1)
    d = Node('D', 2)
    e = Node('E', 0)

    # Add neighbors (node, cost)
    a.add_neighbor(b, 1)
    a.add_neighbor(c, 4)
    b.add_neighbor(d, 2)
    c.add_neighbor(e, 3)
    d.add_neighbor(e, 1)

    # Perform A* Search
    path = a_star_search(a, e)
    if path:
        print("Path found:", " -> ".join(path))
```

OUTPUT

```
Expanding node: A
Expanding node: B
Expanding node: C
Expanding node: D
Expanding node: E
Path found: A -> B -> D -> E
```

EXP NO. 5 <u>Backtracking Algorithm for CSP- Graph Coloring</u>

Algorithm

1. **Input**:
   - A graph represented as an adjacency list (a dictionary where each key is a node and its value is a list of neighboring nodes).
   - A list of colors available for coloring the nodes.
2. **Function `solve_csp_backtracking(graph, colors)`**:
   - Initialize a function to solve the graph coloring problem.
3. **Function `is_safe(node, color, assignment)`**:
   - **Input**: A node, a color, and the current assignment of colors to nodes.
   - **Process**:
     1. For each neighbor of the given node:
        - If the neighbor is already in the assignment and its color matches the proposed color:
          - Return `False` (assignment is not safe).
     2. If all neighbors have different colors, return `True` (assignment is safe).
4. **Function `backtrack(assignment)`**:
   - **Input**: A current assignment of colors to nodes.
   - **Process**:
     1. Check if the length of the assignment equals the number of nodes in the graph:
        - If true, return the current assignment (a valid solution has been found).
     2. Select the first unassigned node from the graph.
     3. For each color in the list of colors:
        1. Check if assigning the current color to the unassigned node is safe using `is_safe`.
        2. If safe:
           - Assign the color to the unassigned node.
           - Recursively call `backtrack` with the updated assignment.
           - If the recursive call returns a valid solution, return that solution.
           - If no solution is found, remove the current assignment (backtrack).
     4. If all colors have been tried and no valid assignment is found, return `None`.
5. **Call the `backtrack` function**:
   - Start with an empty assignment: `backtrack({})`.
6. **Output**:
   - If a solution is found, print the assignments of colors to nodes.
   - If no solution is found, print a message indicating that.

Program

```
def solve_csp_backtracking(graph, colors):

  def is_safe(node, color, assignment):
    """Checks if assigning 'color' to 'node' is safe (doesn't violate constraints)."""
    for neighbor in graph.get(node, []):
      if neighbor in assignment and assignment[neighbor] == color:
        return False
    return True

  def backtrack(assignment):
    """Recursive backtracking function."""
    if len(assignment) == len(graph):
      return assignment  # All nodes assigned, found a solution

    unassigned_node = next((node for node in graph if node not in assignment), None)
    if unassigned_node is None:
      return None

    for color in colors:
      if is_safe(unassigned_node, color, assignment):
        assignment[unassigned_node] = color
        result = backtrack(assignment)
        if result is not None:
          return result
        del assignment[unassigned_node]  # Backtrack: remove assignment

    return None  # No solution found for this branch

  return backtrack({})
```

```
# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D', 'E'],
    'D': ['B', 'C', 'E'],
    'E': ['C', 'D'],
}
colors = ['Red', 'Green', 'Blue']

solution = solve_csp_backtracking(graph, colors)
if solution:
  print("Solution found:")
  for node, color in solution.items():
    print(f"Node {node}: Color {color}")
else:
  print("No solution found.")
```

<u>**OUTPUT**</u>

```
Solution found:
Node A: Color Red
Node B: Color Green
Node C: Color Blue
Node D: Color Red
Node E: Color Green
```

EXP NO 6: Local search algorithms for CSP- Cryptarithmetic Problem

```
 TWO  +

 TWO

-----------

FOUR
```

Algorithm

- **Define Variables**:
  - Identify the letters involved in the problem: `T`, `W`, `O`, `F`, `U`, `R`.
  - Define the possible values for each letter as digits from `0` to `9`.
- **Set Up Constraints**:
  - All letters must have different values.
  - The leading digits (`T` and `F`) cannot be zero.
  - Define the addition constraints based on the carries:

    O + O = R + 10 · C10
    C10 + W + W = U + 10 · C100
    C100 + T + T = O + 10 · C1000
    C1000 = F , where C10, C100, and C1000 are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column
- **Generate Permutations**:
  - Use permutations to generate all possible assignments of digits (from `0` to `9`) to the letters.
  - For each permutation, create a mapping of letters to digits.
- **Validate Assignments**:
  - For each generated assignment:
    - Check that all values are distinct (no two letters have the same digit).
    - Ensure that the leading digits `T` and `F` are not zero.
    - Calculate the carry values:
      - Compute `C10`, `C100`, and `C1000` based on the addition constraints.
    - Verify that the calculated values satisfy the addition constraints.
- **Check for Solution**:
  - If a valid assignment that meets all constraints is found:
    - Return the assignment as the solution.
  - If no valid assignment is found after checking all permutations, conclude that no solution exists.

- **Output**:
    - Print the resulting assignment if a solution is found.
    - Otherwise, print a message indicating that no solution was found.

Program

```python
from itertools import permutations

def is_valid_assignment(assignment):
    """Check if the current assignment satisfies the constraints."""
    T = assignment['T']
    W = assignment['W']
    O = assignment['O']
    F = assignment['F']
    U = assignment['U']
    R = assignment['R']

    # Check for distinct values
    if len(set(assignment.values())) != len(assignment):
        return False

    # Leading digit constraints
    if F == 0 or T == 0:
        return False

    # Extract carry values
    C10 = (O + O) // 10
    C100 = (W + W + C10) // 10
    C1000 = (T + T + C100) // 10

    # Check addition constraints
    if (O + O) % 10 != R:
        return False
    if (W + W + C10) % 10 != U:
        return False
    if (T + T + C100) % 10 != O:
        return False
    if C1000 != F:
        return False

    return True

def local_search():
    """Try all possible assignments for the letters."""
    letters = ['T', 'W', 'O', 'F', 'U', 'R']
    for perm in permutations(range(10), len(letters)):
        assignment = dict(zip(letters, perm))
        if is_valid_assignment(assignment):
            return assignment

    return None  # No solution found

# Solve the cryptarithmetic problem
solution = local_search()
if solution:
    print("Solution found:")
    for letter, digit in solution.items():
        print(f"{letter}: {digit}")
else:
    print("No solution found.")
```

OUTPUT

```
Solution found:
T: 7
W: 3
O: 4
F: 1
U: 6
R: 8
```

EXP7. Game playing (adversarial search)

MIN MAX Game Playing with Alpha-Beta Pruning

AIM

Implement MINMAX Game Playing with Alpha-Beta Pruning

<u>ALGORITHM</u>

1. **Function Definition**:

   Define the function `minimax(node, depth, isMaximizingPlayer, alpha, beta)`.

```
function minimax(node, depth, isMaximizingPlayer, alpha,
beta):
```

2. **Base Case**:

   Check if the current `node` is a leaf node (i.e., no children):

   - If it is a leaf node, return the value of the node.

```
if node is a leaf node :
    return value of the node
```

3. **Maximizing Player Logic**:

```
if isMaximizingPlayer :
    bestVal = -INFINITY
    for each child node :
        value = minimax(node, depth+1, false, alpha, beta)
        bestVal = max( bestVal, value)
        alpha = max( alpha, bestVal)
        if beta <= alpha:
            break
    return bestVal
```

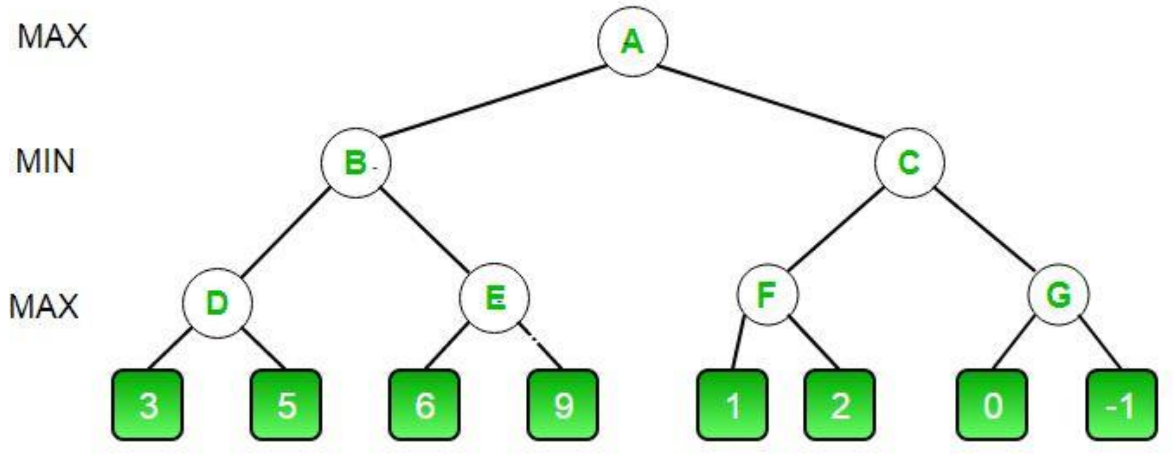
4. `else :` **Minimizing Player Logic**:

```
    bestVal = +INFINITY
    for each child node :
        value = minimax(node, depth+1, true, alpha, beta)
        bestVal = min( bestVal, value)
        beta = min( beta, bestVal)
        if beta <= alpha:
            break
    return bestVal
```

5. **Function Invocation**:

- To start the algorithm, call the `minimax` function with the following parameters:
  - The root node of the game tree.

- `depth` set to `0`.
- `isMaximizingPlayer` set to `True` (indicating it's the maximizing player's turn).
- `alpha` set to `-INFINITY`.
- `beta` set to `+INFINITY`.



PROGRAM

```
# Python3 program to demonstrate

# working of Alpha-Beta Pruning


# Initial values of Alpha and Beta

MAX, MIN = 1000, -1000


# Returns optimal value for current player

#(Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer,

            values, alpha, beta):


    # Terminating condition. i.e

    # leaf node is reached

    if depth == 3:

        return values[nodeIndex]
```

```
if maximizingPlayer:

   best = MIN

   # Recur for left and right children
   for i in range(0, 2):

      val = minimax(depth + 1, nodeIndex * 2 + i,
                    False, values, alpha, beta)
      best = max(best, val)
      alpha = max(alpha, best)

      # Alpha Beta Pruning
      if beta <= alpha:
     print("best value of max player-->",best)

         break

   return best

else:
   best = MAX

   # Recur for left and
   # right children
   for i in range(0, 2):

      val = minimax(depth + 1, nodeIndex * 2 + i,
                     True, values, alpha, beta)
      best = min(best, val)
      beta = min(beta, best)
```

```
            # Alpha Beta Pruning
            if beta <= alpha:
                break
           print("best value of min player-->",best)


        return best


# Driver Code
if __name__ == "__main__":


    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

<u>RESULT</u>

The above program has been successfully executed and output obtained is verified.

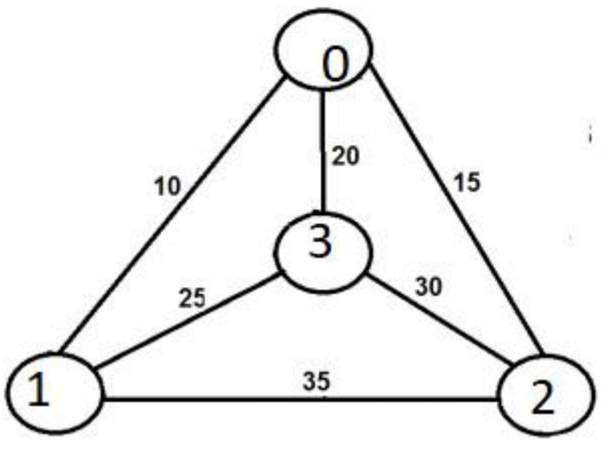
OUTPUT

```
The optimal value is : 5
```

EXP8. TRAVELLING SALESMAN PROBLEM USING HILL CLIMBING

AIM

Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once .



ALGORITHM

1. **Input**:

- A distance matrix representing the distances between each pair of cities.

2. **Initialization**:

- Let `num_cities` be the number of cities (size of the distance matrix).
- Create an initial tour (a list of cities) by generating a list from 1 to `num_cities` (inclusive).
- Randomly shuffle this initial tour to start from a random point.

3. **Calculate Initial Tour Length**:

- Define a function `calculate_tour_length(tour, distance_matrix)`:
    - Initialize a variable `length` to 0.
    - Loop through each city in the tour:
        - For each city, add the distance to the next city in the tour (considering the last city connects back to the first).
    - Return the total `length`.

4. **Hill Climbing Process**:

- Set `current_length` to the length of the initial tour.
- Define a loop that will iterate a specified number of times (e.g., 1000 iterations):
    1. **Generate Neighbor**:
        - Define a function `generate_neighbor(tour)`:
            - Create a copy of the current tour.

        - Randomly select two different indices in the tour.
        - Swap the cities at these indices to create a neighboring tour.
        - Return the neighbor.
2. **Calculate Neighbor Length**:
    - Use the `calculate_tour_length` function to compute the length of the neighboring tour.
3. **Compare Lengths**:
    - If the `neighbor_length` is shorter than `current_length`:
        - Update `current_tour` to the neighbor.
        - Update `current_length` to `neighbor_length`.

5.Output:

- After completing the iterations, the algorithm will have found a potentially improved tour.
- Return the best `current_tour` and its corresponding `current_length`.

6.Display Results:

- Print the best tour and its total distance.

PROGRAM

```
import random

def calculate_tour_length(tour, distance_matrix):
    length = 0
    for i in range(len(tour)):
        length += distance_matrix[tour[i]][tour[(i + 1) % len(tour)]]
    return length

def generate_neighbor(tour):
    neighbor = tour[:]
    # Swap two cities to create a neighbor
    i, j = random.sample(range(len(tour)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

def hill_climbing_tsp(distance_matrix, iterations=1000):
    # Create an initial random tour
    num_cities = len(distance_matrix)
    current_tour = list(range(num_cities))
    random.shuffle(current_tour)
    current_length = calculate_tour_length(current_tour,
distance_matrix)

    for _ in range(iterations):
        neighbor = generate_neighbor(current_tour)
        neighbor_length = calculate_tour_length(neighbor,
distance_matrix)

        if neighbor_length < current_length:
            current_tour = neighbor
            current_length = neighbor_length

    return current_tour, current_length

# Example usage
if __name__ == "__main__":
    # Distance matrix example (symmetric)
    distance_matrix = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]

    best_tour, best_length = hill_climbing_tsp(distance_matrix)
    print("Best tour:", best_tour)
    print("Total distance:", best_length)
```

RESULT

The above program has been successfully executed and output obtained is verified.

OUTPUT

```
Best tour: [2, 0, 1, 3]
Total distance: 80
```

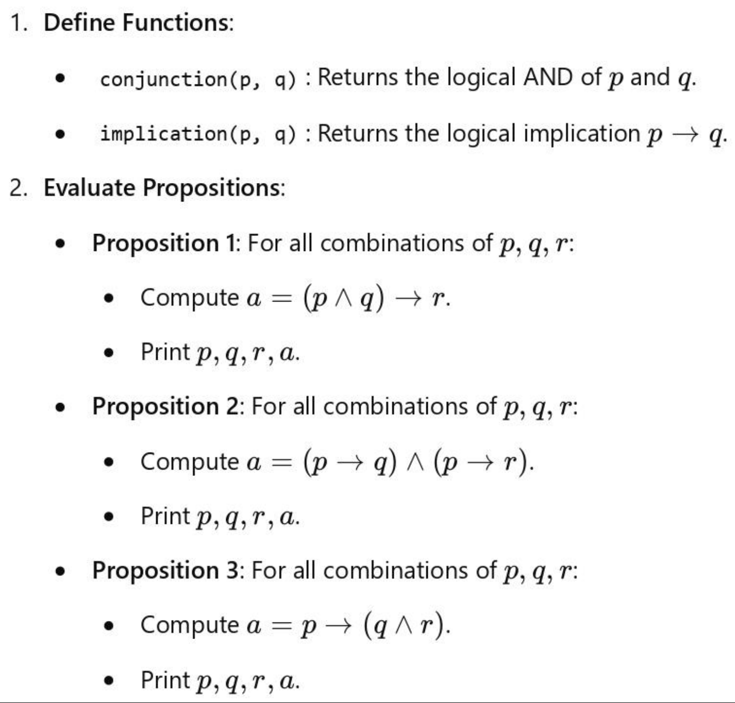
Exp.No. 9 Building propositional logic and display the inference.

AIM:

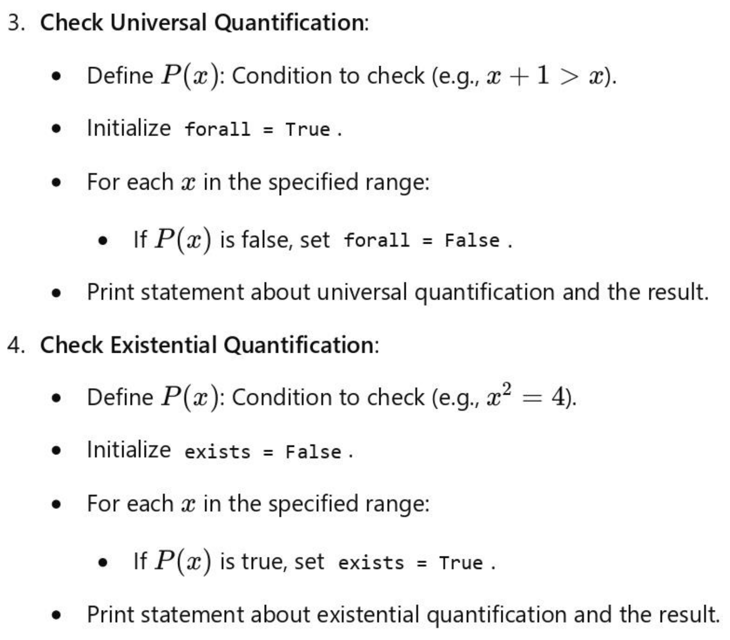
To implement propositional logic inferences for AI tasks

1. (p∧q)⟹r
2. (p⟹q)∧(p⟹r)
3. p⟹(q∧r)
4. ∀x,x+1>x
5. ∃x,x2=4

<u>ALGORITHM</u>



1. **Define Functions**:
    - `conjunction(p, q)` : Returns the logical AND of p and q.
    - `implication(p, q)` : Returns the logical implication $p \rightarrow q$.
2. **Evaluate Propositions**:
    - **Proposition 1**: For all combinations of p, q, r:
        - Compute $a = (p \land q) \rightarrow r$.
        - Print p, q, r, a.
    - **Proposition 2**: For all combinations of p, q, r:
        - Compute $a = (p \rightarrow q) \land (p \rightarrow r)$.
        - Print p, q, r, a.
    - **Proposition 3**: For all combinations of p, q, r:
        - Compute $a = p \rightarrow (q \land r)$.
        - Print p, q, r, a.



3. **Check Universal Quantification**:
    - Define $P(x)$: Condition to check (e.g., $x + 1 > x$).
    - Initialize `forall = True` .
    - For each x in the specified range:
        - If $P(x)$ is false, set `forall = False` .
    - Print statement about universal quantification and the result.
4. **Check Existential Quantification**:
    - Define $P(x)$: Condition to check (e.g., $x^2 = 4$).
    - Initialize `exists = False` .
    - For each x in the specified range:
        - If $P(x)$ is true, set `exists = True` .
    - Print statement about existential quantification and the result.

PROGRAM

```
print("PROPOSITION 1")


def conjunction(p, q):
        return p and q


def implication(p, q):
        return not p or q
print("p    q    r    a")
for p in [True, False]:
        for q in [True, False]:
          for r in [True, False]:
            a = implication(conjunction(p, q), r)
            print(p, q, r, a)
print("********************")
print("PROPOSITION 2")


print("p    q    r    a")
for p in [True, False]:
        for q in [True, False]:
          for r in [True, False]:
            a = conjunction(implication(p, q), implication(p,r))
            print(p, q, r, a)
print("********************")
print("PROPOSITION  3")



print("p    q    r    a")
for p in [True, False]:
        for q in [True, False]:
          for r in [True, False]:
            a = implication(p, conjunction(q,r))
            print(p, q, r, a)
print("********************")
print("PROPOSITION 4")

stmt = "For all x, P(x)."
def P(x):
        return x + 1 > x

forall = True
for x in [-2, -1, 0, 1, 2]:
        # check if at least 1 is false
        if P(x) == False:
          forall = False

print(stmt)
print(forall)
print("********************")
print("PROPOSITION 5")

stmt = "There exists an x, such that P(x)."
def P(x):
        return x**2 == 4

exists = False
for x in [-2, -1, 0, 1, 2]:
        # check if at least 1 is true
        if P(x):
          exists = True
print(stmt)
print(exists)
```

OUTPUT

```
PROPOSITION 1
p    q    r    a
True True True True
True True False False
True False True True
True False False True
False True True True
False True False True
False False True True
False False False True
********************
PROPOSITION 2
p    q    r    a
True True True True
True True False False
True False True False
True False False False
False True True True
False True False True
False False True True
False False False True
********************
PROPOSITION  3
p    q    r    a
True True True True
True True False False
True False True False
True False False False
False True True True
False True False True
False False True True
False False False True
********************
PROPOSITION 4
For all x, P(x).
True
********************
PROPOSITION 5
There exists an x, such that P(x).
True
```

EXP NO 10: propositional logic inference for AI Inference Rules

AIM

Write a python program to implement propositional logic inferences for AI Inference Rules

1. **Modus Ponens**: From P and $P \rightarrow Q$, infer Q.
2. **Modus Tollens**: From $\neg Q$ and $P \rightarrow Q$, infer $\neg P$.
3. **Disjunctive Syllogism**: From $P \vee Q$ and $\neg P$, infer Q.

ALGORITHM

1. **Define `Proposition` Class**:
   - Attributes:
     - `name` : Description of the proposition.
     - `value` : Boolean truth value (True/False).
   - Methods:
     - `negate()` : Returns the negation of the proposition.
     - `__or__(other)` : Returns the disjunction of two propositions.
     - `__and__(other)` : Returns the conjunction of two propositions.
2. **Define Inference Functions**:
   - **Modus Ponens**: If P is True and $P \rightarrow Q$ is defined, return Q.
   - **Modus Tollens**: If $\neg Q$ is True and $P \rightarrow Q$ is defined, return $\neg P$.
   - **Disjunctive Syllogism**: If $P \vee Q$ is True and $\neg P$ is True, return Q.
3. **Initialize Propositions**:
   - Create `in_class` as True ("You are in the class").
   - Create `grade` as False ("You will get a grade").
   - Create `not_grade` using the `negate()` method.
4. **Apply Inference Rules**:
   - Call inference functions and display results:
     - Modus Ponens with `in_class` and the implication.
     - Modus Tollens with `not_grade` .
     - Disjunctive Syllogism with the disjunction of `in_class` and `grade` .

PROGRAM

```
class Proposition:
    def __init__(self, name, value=None):
        self.name = name
        self.value = value

    def negate(self):
        return Proposition(f"¬{self.name}", not self.value)

    def __or__(self, other):
        return Proposition(f"({self.name} ∨ {other.name})", self.value
or other.value)

    def __and__(self, other):
        return Proposition(f"({self.name} ∧ {other.name})", self.value
and other.value)

    def __repr__(self):
        return self.name

def modus_ponens(p, p_implies_q):
    if p.value and p_implies_q.value is not None:
        return Proposition("You will get a grade", True)
    return Proposition("You will get a grade", False)

def modus_tollens(not_q, p_implies_q):
    if not_q.value and p_implies_q.value is not None:
        return Proposition("You are not in the class", True)
    return Proposition("You are not in the class", False)

def disjunctive_syllogism(p_or_q, not_p):
    if not_p.value and p_or_q.value is not None:
        return Proposition("You will get a grade", True)
    return Proposition("You will get a grade", False)

# Sample input based on the statement
# Define propositions
in_class = Proposition("You are in the class", True)  # You are in the
class
grade = Proposition("You will get a grade", False)  # Initial
assumption about getting a grade
not_grade = grade.negate()  # ¬(You will get a grade)
class_implies_grade = Proposition("You are in the class → You will get
a grade", None)  # P → Q (unknown)

# Applying Modus Ponens
result_mponens = modus_ponens(in_class, class_implies_grade)
print("Modus Ponens Result:", result_mponens)

# Applying Modus Tollens
result_mtollens = modus_tollens(not_grade, class_implies_grade)
print("Modus Tollens Result:", result_mtollens)

# Applying Disjunctive Syllogism
```

```python
class_or_grade = in_class | grade  # You are in the class ∨ You will get a grade
result_ds = disjunctive_syllogism(class_or_grade, in_class.negate())
print("Disjunctive Syllogism Result:", result_ds)
```

OUTPUT

```
Modus Ponens Result: You will get a grade
Modus Tollens Result: You are not in the class
Disjunctive Syllogism Result: You will get a grade
```

EXP NO: 11 PROPOSITIONAL LOGIC INFERENCES

AIM:
To familiarize and understand the use of propositional_logic

If it didn't rain, Harry visited Hagrid today.

Harry visited Hagrid or Dumbledore today, but not both.

Harry visited Dumbledore today.

Represent these knowledge in knowledge base

Check whether "harry visited hagrid"

Check whether "it rained today"

ALGORITHM

Algorithm

1. **Initialize Environment**:
   - Import necessary libraries:
     - `from sympy import symbols, And, Or, Not, Implies, satisfiable`
2. **Define Propositions**:
   - Create symbols to represent the relevant propositions:
     - `rain`: Represents whether it is raining.
     - `hagrid`: Represents whether Harry visited Hagrid.
     - `dumbledore`: Represents whether Harry visited Dumbledore.
3. **Construct Knowledge Base**:
   - Initialize the knowledge base as a logical conjunction of the following statements:
     1. **If it didn't rain, then Harry visited Hagrid**:
        - Use the implication: `Implies(Not(rain), hagrid)`
     2. **Harry did not visit both Hagrid and Dumbledore**:
        - Use negation and conjunction: `Not(And(hagrid, dumbledore))`
     3. **Harry visited either Hagrid or Dumbledore**:
        - Use disjunction: `Or(hagrid, dumbledore)`
     4. **Fact: Harry visited Dumbledore**:
        - Simply include: `dumbledore`
   - Combine these statements into the knowledge base using `And()`.
4. **Display Knowledge Base**:
   - Print the constructed knowledge base to visualize the logical structure.
5. **Check Satisfiability of Knowledge Base**:
   - Use the `satisfiable()` function to determine if the knowledge base can hold true for some assignment of truth values.
   - Store the result in a variable (e.g., `is_satisfiable`).
   - Print whether the knowledge base is satisfiable.
6. **Check if Harry Visited Hagrid**:

- Substitute `hagrid` with `True` in the knowledge base:
    - Use `satisfiable(knowledge_base.subs({hagrid: True}))`.
- Store the result in a variable (e.g., `hagrid_check`).
- Print the result of the check.

7. **Check if It Rained Today**:
    - Substitute `rain` with `True` in the knowledge base:
        - Use `satisfiable(knowledge_base.subs({rain: True}))`.
    - Store the result in a variable (e.g., `rain_check`).
    - Print the result of the check.

PROGRAM

```
rom sympy import symbols, And, Or, Not, Implies, satisfiable

# Define the symbols
rain = symbols("rain")          # It is raining
hagrid = symbols("hagrid")      # Harry visited Hagrid
dumbledore = symbols("dumbledore")  # Harry visited Dumbledore

# Create the knowledge base (KB) with the given facts
knowledge_base = And(
    Implies(Not(rain), hagrid),              # If it didn't rain, then
Harry visited Hagrid
    Not(And(hagrid, dumbledore)),            # Harry did not visit both
Hagrid and Dumbledore
    Or(hagrid, dumbledore),                  # Harry visited either
Hagrid or Dumbledore
    dumbledore                                # Fact: Harry visited
Dumbledore
)

# Display the knowledge base
print("Knowledge Base:")
print(knowledge_base)

# Check satisfiability of the knowledge base
is_satisfiable = satisfiable(knowledge_base)
print("\nIs the knowledge base satisfiable?", is_satisfiable)

# Check if "Harry visited Hagrid"
hagrid_check = satisfiable(knowledge_base.subs({hagrid: True}))
print("\nDid Harry visit Hagrid?", hagrid_check)

# Check if "It rained today"
rain_check = satisfiable(knowledge_base.subs({rain: True}))
print("Did it rain today?", rain_check)
```

OUTPUT

```
Knowledge Base:
dumbledore & (dumbledore | hagrid) & (Implies(~rain, hagrid)) &
~(dumbledore & hagrid)

Is the knowledge base satisfiable? {dumbledore: True, rain: True,
hagrid: False}

Did Harry visit Hagrid? False
Did it rain today? {dumbledore: True, hagrid: False}
```

EXP NO: 11 .2 PROPOSITIONAL LOGIC INFERENCES

AIM:
To familiarize and understand the use of propositional_logic

Construct a knowledge base using the following rules:

1. John likes all food.
2. Peanuts, apples, and vegetables are considered food.
3. Anything that is eaten and not killed is food.
4. Anil eats peanuts and is alive.
5. Harry eats everything that Anil eats.

Check : John Likes Peanuts

Algorithm

- **Initialize Environment**:
  - Import necessary libraries:
    - `from sympy import symbols, And, Or, Implies, satisfiable`
- **Define Propositions**:
  - Create symbols representing the relevant concepts:
    - `john_likes`: Represents whether John likes peanuts.
    - `peanuts`: Represents the food item peanuts.
    - `alive`: Represents the alive status of Anil.
    - `food`: Represents the general concept of food.
    - `eats`: Represents the action of eating.
    - `anil`: Represents Anil.
    - `harry`: Represents Harry.
- **Construct Knowledge Base**:
  - Combine logical statements into the knowledge base (`knowledge_base`) using `And()`:
    1. **John likes all food**:
       - Use implication: `Implies(food, john_likes)`
    2. **Define food items**:
       - Use disjunction: `Or(peanuts, symbols("apple"), symbols("vegetable"))`
    3. **Define what constitutes food**:
       - Use implication: `Implies(And(eats, alive), food)`
    4. **Specify Anil's eating habits**:
       - State: `And(eats.subs({anil: True, peanuts: peanuts}), alive.subs({anil: True}))`
    5. **Harry's eating habits**:

        - Use implication: `Implies(eats.subs({anil: True, symbols("x"): symbols("x")}), eats.subs({harry: True, symbols("x"): symbols("x")}))`

- **Check John's Preference for Peanuts**:

  - Use the `satisfiable()` function to check if the knowledge base supports that John likes peanuts:
    - Substitute `peanuts` with `True` in the knowledge base.
  - Store the result in a variable (e.g., `john_likes_peanuts_check`).

PROGRAM

```
from sympy import symbols, And, Or, Not, Implies, satisfiable

# Define the symbols
john_likes = symbols("john_likes(peanuts)")  # John likes peanuts
peanuts = symbols("peanuts")                  # Peanuts
alive = symbols("alive")                      # Alive status
food = symbols("food")                        # Food status
eats = symbols("eats")                        # Eating action
anil = symbols("anil")                        # Anil
harry = symbols("harry")                      # Harry

# Create the knowledge base (KB)
knowledge_base = And(
    Implies(food, john_likes),                          # John likes all food
    Or(peanuts, symbols("apple"), symbols("vegetable")),  # Apple and vegetable are food
    Implies(And(eats, alive), food),                    # Anything eaten and alive is food
    And(eats.subs({anil: True, peanuts: peanuts}), alive.subs({anil: True})),  # Anil eats peanuts and is alive
    Implies(eats.subs({anil: True, symbols("x"): symbols("x")}), eats.subs({harry: True, symbols("x"): symbols("x")}))  # Harry eats everything that Anil eats
)

# Display the knowledge base
print("Knowledge Base:")
print(knowledge_base)

# Check if "John likes peanuts"
john_likes_peanuts_check = satisfiable(knowledge_base.subs({peanuts: True}))
print("\nDoes John like peanuts?", john_likes_peanuts_check)
```

OUTPUT

```
Knowledge Base:
alive & eats & (Implies(food, john_likes(peanuts))) & (apple | peanuts
| vegetable) & (Implies(alive & eats, food))

Does John like peanuts? {eats: True, alive: True, food: True,
john_likes(peanuts): True}
```

Experiment No:-12 IMPLEMENTATION OF KNOLEDGE REPRESENTATION

AIM:
To familiarize and understand the use of propositional_logicknowledge_representation

Question:-

Create a knowledge base with the following facts and find out the answer for the queries listed below

1.Noor likes sausage ,it's food type is meat and the flavour of meat is savoury.

2. Dmitry likes cookies ,it's food type is dessert and the flavour of cookie is sweet.

3. Aisel likes lemonade,it's food type is juice and the flavour of juice is sweet.

4. Mellissa likes pasta ,it's food type is cheese and the flavour of cheese is savoury.

I) Recommend dishes to persons based on taste preferences.
II) Find out the food flavour of a dish liked by a person
III) Check whether Noor likes pasta.

ALGORITHM

- **Install Library:**
  - Use `pip` to install the `pytholog` library.
- **Initialize Knowledge Base:**
  - Create a new knowledge base named "flavor".
- **Define Facts:**
  - Add the following facts to the knowledge base:
    - Likes: `likes(noor, sausage)`, `likes(melissa, pasta)`, etc.
    - Food types: `food_type(gouda, cheese)`, `food_type(ritz, cracker)`, etc.
    - Flavors: `flavor(sweet, dessert)`, `flavor(savory, meat)`, etc.
- **Define Rules:**
  - Rule for food flavor:
    - `food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z)`
  - Rule for dishes to like:
    - `dish_to_like(X, Y) :- likes(X, L), food_type(L, T), flavor(F, T), food_flavor(Y,F), neq(L, Y)`
- **Perform Queries:**

- Query if `noor` likes `sausage`.
- Query if `noor` likes `pasta`.
- Query for foods with a sweet flavor.
- Query for dishes `noor` might like.
- Query for foods with a savory flavor.

- **Output Results:**
    - Print the results of the flavor and dish queries.

PROGRAM

```
!pip install pytholog

import pytholog as pl

new_kb = pl.KnowledgeBase("flavor")

new_kb(["likes(noor, sausage)",

 "likes(melissa, pasta)",

 "likes(dmitry, cookie)",

 "likes(nikita, sausage)",

 "likes(assel, limonade)",

 "food_type(gouda, cheese)",

 "food_type(ritz, cracker)",

 "food_type(steak, meat)",

 "food_type(sausage, meat)",

 "food_type(limonade, juice)",

 "food_type(mojito, juice)",

 "food_type(cookie, dessert)",

 "flavor(sweet, dessert)",

 "flavor(savory, meat)",

 "flavor(savory, cheese)",

 "flavor(sweet, juice)",

  "food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z)",

 "dish_to_like(X, Y) :- likes(X, L), food_type(L, T), flavor(F, T), food_flavor(Y,F), neq(L, Y)"])
```

```
new_kb.query(pl.Expr("likes(noor, sausage)"))

new_kb.query(pl.Expr("likes(noor, pasta)"))
```

```
print(new_kb.query(pl.Expr("food_flavor(What, sweet)")))
print(new_kb.query(pl.Expr("dish_to_like(noor, What)")))
print(new_kb.query(pl.Expr("food_flavor(What,savory)")))
```

OUTPUT

```
['Yes']


['No']

[{'What': 'cookie'}, {'What': 'limonade'}, {'What': 'mojito'}]
[{'What': 'gouda'}, {'What': 'steak'}]
[{'What': 'gouda'}, {'What': 'sausage'}, {'What': 'steak'}]
```

AIL 333 AI ALGORITHMS LAB
VIVA QUESTIONS

1. **Artificial intelligence** is a wide-ranging branch of computer science concerned with building smart machines capable of performing tasks that typically require human intelligence.
2. The **Turing test** was developed by Alan Turing(A computer scientist) in 1950. He proposed that the "Turing test is used to determine whether or not a computer(machine) can think intelligently like humans
3. **TOTAL TURING TEST**- To pass the total Turing Test, the computer will need computer vision to perceive objects, and robotics to manipulate objects and move about.
4. **Requirements for AI**
   - NATURAL LANGUAGE PROCESSING
   - To enable it to communicate successfully
   - KNOWLEDGE REPRESENTATION
   - Knowledge representation to store what it knows or hears;
   - AUTMATED REASONING
   - Automated reasoning to use the stored information to answer questions and to draw new conclusions
   - MACHINE LEARNING
   - machine learning to adapt to new circumstances and to detect and extrapolate patterns.
   - COMPUTER VISION
   - Computer vision to perceive objects
   - ROBOTICS
   - Robotics to manipulate objects and move about

5. An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**
6. **An intelligent agent** is an autonomous entity which act upon an environment using sensors and actuators for achieving goals. An intelligent agent may learn from the environment to achieve their goals. A thermostat is an example of an intelligent agent.
7. A **rational agent** is an agent which has clear preference, models uncertainty, and acts in a way to maximize its performance measure with all possible actions.
8. **Structure of an AI Agent**
   Agent = Architecture + Agent program
9. **PEAS Representation**
   PEAS System is used to categorize similar agents together. The PEAS system delivers the performance measure with respect to the environment, actuators, and sensors of the respective agent. Most of the highest performing agents are Rational Agents.

PEAS stands for a *Performance measure, Environment, Actuator, Sensor*.

10. Types of Environments in AI

- Fully Observable vs Partially Observable
- Deterministic vs Stochastic
- Competitive vs Collaborative
- Single-agent vs Multi-agent
- Static vs Dynamic
- Discrete vs Continuous
- Episodic vs Sequential
- Known vs Unknown

11. **Types of Agent Programs**

Four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

1. Simple reflex agents;
2. Model-based reflex agents;
3. Goal-based agents; and
4. Utility-based agents

12. Problem formulation

Components to formulate the associated problem

Initial State: This state requires an initial state for the problem which starts the AI agent towards a specified goal. In this state new methods also initialize problem domain solving by a specific class.

Action: This stage of problem formulation works with function with a specific class taken from the initial state and all possible actions done in this stage.

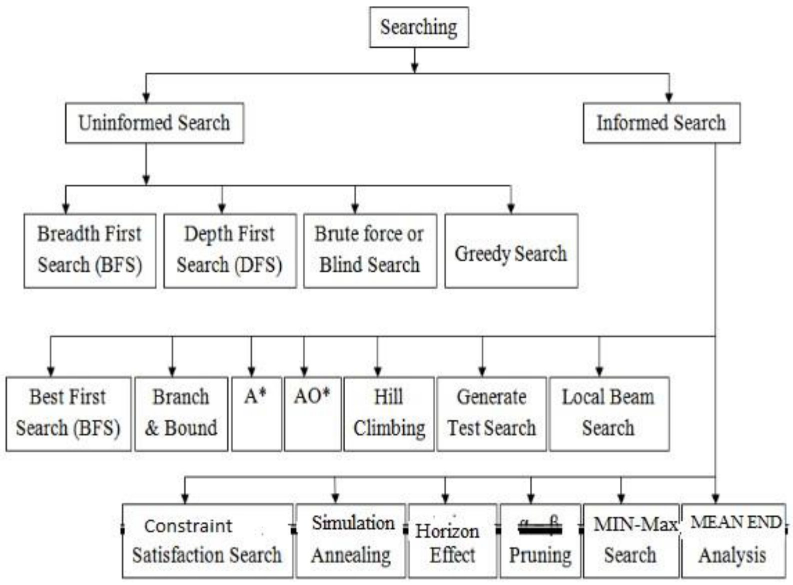
Transition: This stage of problem formulation integrates the actual action done by the previous action stage and collects the final stage to forward it to their next stage.

Goal test: This stage determines that the specified goal achieved by the integrated transition model or not, whenever the goal achieves stop the action and forward into the next stage to determines the cost to achieve the goal.

Path costing: This component of problem-solving numerical assigned what will be the cost to achieve the goal. It requires all hardware software and human working cost.

13. **Uninformed Search Strategies**

In this there is no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. They that do not take into account the location of the goal. These algorithms ignore where they are going until they find a goal and report success.



14.

15. Breadth-First search

The root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on. All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. FIFO Queue is used.

16. Uniform-cost search

Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* g(n).

17. **Depth-first search**

Depth-first search always expands the *deepest* node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.LIFO Stack is used.

18. Depth-limited search

Depth-limited search is depth-first search with a predetermined depth limit l nodes at depth l are treated as if they have no successors.

19. Iterative deepening depth-first search

Iterative deepening search is often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node.

20. **INFORMED (HEURISTIC) SEARCH STRATEGIES**

In Informed search problem-specific knowledge beyond the definition of the problem itself can find solutions more efficiently than an uninformed strategy.

21. **BEST-FIRST SEARCH**

Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, f(n). The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.

22. **Greedy best-first search**
Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. It evaluates nodes by using just the heuristic function; that is, f(n) = h(n).

23. A* search: Minimizing the total estimated solution cost
It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:
f(n) = g(n) + h(n)
Since g(n) gives the path cost from the start node to node n, and
h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.

24. Heuristics Function
The performance of heuristic search algorithms depends on the quality of the heuristic function. One can sometimes construct good heuristics by relaxing the problem definition, by storing precomputed solution costs for sub problems in a pattern database, or by learning from experience with the problem class.
25. **ADVERSARIAL SEARCH**
In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us. Game Playing
26. PRUNING
We need an algorithm to find optimal move and choosing a good move when time is limited.
Pruning allows us to ignore portions of the search tree that make no difference to the final choice, and heuristic **evaluation functions** allow us to approximate the true utility of a state without doing a complete search
27. **MIN- MAX**
MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following elements:
28. Game tree
The initial state, ACTIONS function, and RESULT function define the **game tree** for the game a tree where the nodes are game states and the edges are moves. From the initial state, MAX has nine possible moves.
29. **ALPHA BETA PRUNING**
In alpha beta pruning algorithm it prunes away branches that cannot possibly influence the final decision. Alpha– beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.
30. **Constraint Satisfaction Problem**
CSP problem is solved when each variable has a value that satisfies all the constraints on the variable
Defining Constraint Satisfaction Problems
A constraint satisfaction problem consists of three components, X, D, and C:
 X is a set of variables, {X1,...,Xn}.

 D is a set of domains, {D1,...,Dn}, one for each variable.

 C is a set of constraints that specify allowable combinations of values.

Example problem: Map coloring, Job-shop scheduling,
Timetabling

31. Types of constraints in CSP

1. **Unary constraint-** which restricts the value of a single variable
◦ Eg, map-coloring problem it could be the case that South Australians won't tolerate the color green; we can express that with the unary constraint
2. **binary constraint-** relates two variables
◦ A binary CSP is one with only binary constraints; it can be represented as a constraint graph
3. We can also describe higher-order constraints, such as asserting that the value of Y is between X and Z, with the ternary constraint Between(X, Y, Z).

4. **global constraint-** A constraint involving an arbitrary number of variables
◦ Eg, In Sudoku problems all variables in a row or column must satisfy an Alldiffff , which says that all of the variables involved in the constraint must have different values

32. Constraint hypergraph

The constraints can be represented in a **constraint hypergraph**. A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints.

33. **CONSTRAINT PROPAGATION: INFERENCE IN CSPS**

Local consistency

If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph

Node consistency

Avariable with the values in the variable's domain satisfy the variable's unary constraints. Eg: variant of the Australia map-coloring problem with South Australians dislike green, the variable SA starts with domain {red, green, blue}, and we can make it node consistent by eliminating green, leaving SA with the reduced domain {red, blue}.

A network is node-consistent if every variable in the network is node-consistent**.** It is always possible to eliminate all the unary constraints in a CSP by running node consistency.

Arc consistency

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. Xi is arc-consistent with respect to another variable Xj. If for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi, Xj). A network is arc-consistent if every variable is arc consistent with every other variable.

Path consistency

Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

K-consistency
Stronger forms of propagation can be defined with the notion of k**-consistency**. A CSP is k-consistent if, for any set of k − 1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.

Global constraints
Global constraint is one involving an arbitrary number of variables but not necessarily all variables.

34. **Backtracking search**
It is a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution.

35. **Minimum-remaining-values (MRV) heuristic**
The idea of choosing the variable with the fewest "legal" value. A.k.a. "**most constrained variable**" or "fail-first" heuristic, it picks a variable that is most likely to cause a failure soon thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately avoiding pointless searches through other variables.
36. **Least Constraining Value**
Once a variable has been selected, the algorithm must decide on the order in which to examine its values. **Least-constraining-value** prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph..
37. **Forward checking**
This is one of the simplest forms of inference. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.
38. The **backjumping** method backtracks to the *most recent* assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.
39. **How to solve a tree-structure CSP:**
Pick any variable to be the root of the tree; Choose an ordering of the variable such that each variable appears after its parent in the tree. (**topological sort**).
40. **Logical Agents**
The idea is that an agent can represent knowledge of its world, its goals and the current situation by sentences in logic and decide what to do by inferring that a certain action or course of action is appropriate to achieve its goals.
41.
Knowledge-Based Agents

 Central component of a Knowledge-Based Agent is a Knowledge-Base

 It is a set of sentences in a formal language

 Sentences are expressed using a knowledge representation language

41. **Inference and Entailment**

Inference is a procedure that allows new sentences to be derived from a knowledge base. Understanding inference and entailment: think of Set of all consequences of a KB as a haystack, α as the needle, Entailment is like the needle being in the haystack and Inference is like finding it.

42. **Propositional Logic: A Very Simplest Logic**

defines the allowable sentences or propositions.

43. Inference

Modus PonensModus Tolens

44. **Demerit of Propositional Logic**

Propositional logic can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power.

45. First-Order Logic (FOL)

It is an extension to propositional logic. FOL is sufficiently expressive to represent the natural language statements in a concise way. First-order logic is also known as Predicate logic or First-order predicate logic.

46. **Resolution**

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. Resolution is a single inference rule which can efficiently operate on the conjunctive normal form or clausal form.

47. **Unification**

The process of finding a substitution for predicate parameters is called unification. Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process. It takes two literals as input and makes them identical using substitution. We need to know: that 2 literals can be matched. The substitution is that makes the literals identical. There is a simple algorithm called the unification algorithm that does this.

48. **Inference engine**

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. Inference engine commonly proceeds in two modes, which are:

 Forward chaining

 Backward chaining

49. Backward Chaining

Back Chaining considers the item to be proven a goal. Find a rule whose head is the goal (and bindings). Apply bindings to the body, and prove these (subgoals) in turn. If you prove all the subgoals, increasing the binding set as you go, you will prove the item.

50. Forward Chaining

Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached. The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

ADDED QUESTIONS

1. What is artificial intelligence?

It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence. AI doesn't have to confine itself to methods that are biologically observable.

2. What is intelligence?

Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals and some machines.

3. What is the Turing Test?

The Turing test is a one-sided test. A machine that passes the test should certainly be considered intelligent, but a machine could still be considered intelligent without knowing enough about humans to imitate a human.

4. What are the applications of AI?

● Game playing. ● Heuristic classification. ● Expert systems ● Computer vision ● Understanding natural language ● Speech recognition

5. What is game playing?

You can buy machines that can play master level chess for a few hundred dollars. They play well against people mainly through brute force computation looking at hundreds of thousands of positions. To beat a world champion by brute force and known reliable heuristics requires being able to look at 200 million positions per second.

6. What do you mean by an Expert system?

An expert system, also know as a knowledge based system. It is a computer program that contains some of the subject-specific knowledge, and contains the knowledge and analytical skills of one or more human experts. The class of program was first developed by researchers in artificial intelligence during the 1960s and 1970s and applied commercially throughout the 1980s.

7. What is an agent?

The term "agent" describes a software subtraction, an idea, or a concept, similar to OOP terms such as methods, functions and objects. The concept of an agent provides a convenient and powerful way to describe a complex software entity that is capable of acting with a certain degree of autonomy in order to accomplish tasks on behalf of its user. But unlike objects, which are defined in terms of methods and attributes, an agent is defined in terms of its behaviour.

8. What do you mean by a Heuristic?

It can be either be any algorithm that gives up finding the optimal solution for an improvement in run time. It can be a function that estimates the cost of the cheapest path from one node to another.

9. What is DFS?

Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure or graph. It is an uninformed search algorithm where the deepest non-terminal node is expanded first.

10. What is A* algorithm?

A* (pronounced "A star" is a graph/ tree search algorithm that finds a path from a given

initial node to a given goal node (or one passing a given goal test). It employs a "heuristic estimate" h(x) that ranks each node x by an estimate of the best route that goes through that node. The A* algorithm is therefore an example of best-first search.

11. What do you mean by BFS?

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. It is an uninformed search algorithm where the shallowest node in the search tree is expanded first. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal

12. A*: A search algorithm to find the shortest path through a search space to a goal state using a heuristic. See 'search', 'problem space', admissibility' and 'heuristic'.

13. Admissibility: An admissible search algorithm is one that is guaranteed to find the optimal path fron the start node to a goal node, if one exists. In A* search, an admissible heuristic is one that never overestimated the distance remaining from the current node to the goal.

14. Agent "Anything that can be viewed a perceiving its environment through sensors and acting upon that environment through effectors." [Russel, Norvig 1995]

15. Alpha-Beta Prunning: A method of limiting search in the MiniMax algorithm

16. Backward Chaining: In a logic system, reasoning from a query to the data. See forward chaining

17. Evaluation Function: A function applied to a game state to generate a guess as to who is winning. Used by Minimax when the game tree is too large to be searched exhaustively.

18. Heuristic: The dictionary defines it as a method that serves as an aid to problem solving. It si sometimes defined as any 'rule of thumb'. Technically, a heuristic is a function that takes a state as input and output a value for that state often as a guess of how far away that state is from the goal state. See also: Admissibility, Search.

18. Iterative Deepening: It is an uninformed search that combines good properties of depth-first and breadth-first search. The ideas of iterative deepening applied to A*.

19. Machine Learning: A field of AI concerned with programs that learn. It includes reinforcement Learning and Neural Networks among many other fields.

20. Min Max: It is an algorithm for game playing in games with perfect information

21.Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value. One of the widely discussed examples of Hill climbing algorithm is **Traveling-salesman Problem** in which we need to minimize the distance traveled by the salesman.

22. **Genetic Algorithms(GAs**) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

23. Expert System

An expert system is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert. It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries. Example: DENDRAL, MYCIN

Components of Expert System

- **User Interface**
- **Inference Engine**
- **Knowledge Base**

24. Production System in AI

A production system (popularly known as a production rule system) is a kind of cognitive architecture that is used to implement search algorithms and replicate human problem-solving skills. This problem-solving knowledge is encoded in the system in the form of little quanta popularly known as productions. It consists of two components: rule and action.

25. Heuristic Search

A heuristic approach in AI is a way of problem-solving that employs intelligent strategies to find solutions. It seeks the most optimal solution but does not guarantee it, instead opting for the best available option at any given time.

- A* search
- Hill Climbing
- Best First search

****** BEST OF LUCK**********