Kartik Padave

A022

70362019039

15/03/2022

AI Practical 1

Aim: Introduction to Prolog

Prolog is a programming language created for artificial intelligence programs. Prolog is made as a declarative programming language i.e., the program logic is expressed in terms of relations, represented as facts and rules. It has many free and commercial implementations available. The language has been used for theorem proving, expert systems, natural language processing and many more. Modern Prolog supports creation of graphical user interfaces, as well as administrative and networked applications.

Loading code in Prolog is called consulting. It can be done by entering queries at the Prolog prompt ?-. If no solution is found, Prolog writes no and if a solution exists, then it is printed. In case of multiple solutions, a semi-colon ;, can be used at end of query. Prolog also provides guidelines on good programming practice to improve code efficiency, readability, and maintainability.

Frameworks exist which can bridge between Prolog and other languages:

- The LPA Intelligence Server allows the embedding of LPA Prolog for Windows within C, C#, C++, Java, VB, Delphi, .Net, Lua, Python, and other languages. It exploits the dedicated string datatype which LPA Prolog provides
- The Logic Server API allows both the extension and embedding of Prolog in C, C++, Java, VB, Delphi, .NET, and any language/environment which can call a .dll or .so. It is implemented for Amzi! Prolog Amzi! Prolog + Logic Server but the API specification can be made available for any implementation.
- JPL is a bi-directional Java Prolog bridge which ships with SWI-Prolog by default, allowing Java and Prolog to call each other (recursively). It is known to have good concurrency support and is under active development.

- InterProlog, a programming library bridge between Java and Prolog, implementing bi-directional predicate/method calling between both languages. Java objects can be mapped into Prolog terms and vice versa. Allows the development of GUIs and other functionality in Java while leaving logic processing in the Prolog layer. Supports XSB, with support for SWI-Prolog and YAP planned for 2013.
- Prova provides native syntax integration with Java, agent messaging and reaction rules. Prova positions itself as a rule-based scripting (RBS) system for middleware. The language breaks new ground in combining imperative and declarative programming.
- PROL An embeddable Prolog engine for Java. It includes a small IDE and a few libraries.
- GNU Prolog for Java is an implementation of ISO Prolog as a Java library (gnu.prolog)
- Ciao provides interfaces to C, C++, Java, and relational databases.
- C#-Prolog is a Prolog interpreter written in (managed) C#. Can easily be integrated in C# programs. Characteristics: reliable and fast interpreter, command line interface, Windows-interface, builtin DCG, XML-predicates, SQL-predicates, extendible. The complete source code is available, including a parser generator that can be used for adding special purpose extensions.
- A Warren Abstract Machine for PHP A Prolog compiler and interpreter in PHP 5.3. A library that can be used standalone or within Symfony2.1 framework which was translated from Stephan Buettcher's work in Java which can be found [here stefan.buettcher.org/cs/wam/index.html]

Prolog has been used in Watson which uses IBM's DeepQA software and the Apache Unstructured Information Management Architecture (UIMA) framework. The system was written using Java, C++, and Prolog, and runs on the SUSE Linux Enterprise Server 11 operating system using Apache Hadoop framework to provide distributed computing. Prolog is used for pattern matching over natural language parse trees and is also used in Low-Code Development Platform GeneXus, which is focused on AI.

Practical 2

Aim: Write a program in Prolog to implement Family Tree

Define:

Relationship is one of the main features that we must properly mention in Prolog. These relationships can be expressed as facts and rules. After that we will see about the family relationships, how we can express family-based relationships in Prolog and see the recursive relationships of the family.

Code:

```
parent(prashant, kartik).
parent(priya, kartik).
parent(prashant, chaitanya).
parent(priya, chaitanya).
parent(vinay, varun).
parent(poonam, varun).
parent(vinay, akshata).
parent(poonam, akshata).
parent(nitin, nishita).
parent(sunita, nishita).
parent(nitin, ruchita).
parent(sunita, ruchita).

male(kartik).
male(chaitanya).
male(prashant).
male(vinay).
male(nitin).
male(varun).

female(priya).
female(akshata).
female(ruchita).
female(nishita).
female(poonam).
female(sunita).

father_of(X, Y):-
    male(X), parent(X, Y).

mother_of(X, Y):-
    female(X), parent(X, Y).

brother_of(X, Y):-
    male(X), parent_of(Z, X), parent_of(Z, Y).
```

```
sister_of(X, Y):-
    female(X), parent(Z, X), parent(Z, Y).
```

Input:

```
parent(prashant, kartik).
parent(priya, kartik).
parent(prashant, chaitanya).
parent(priya, chaitanya).
parent(vinay, varun).
parent(poonam, varun).
parent(vinay, akshata).
parent(poonam, akshata).
parent(nitin, nishita).
parent(sunita, nishita).
parent(nitin, ruchita).
parent(sunita, ruchita).

male(kartik).
male(chaitanya).
male(prashant).
male(vinay).
male(nitin).
male(varun).

female(priya).
female(akshata).
female(ruchita).
female(nishita).
female(poonam).
female(sunita).
```

Output:

```
?- father_of(X, Y).
X = prashant,
Y = kartik ,

?- mother_of(X, Y).
X = priya,
Y = kartik ,

?- brother_of(X, Y).
X = Y, Y = kartik ,

?- sister_of(X, Y).
X = akshata,
Y = varun ,
```

Practical 3

Aim: Write a program to implement Breadth First Search algorithm

Define:

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

Code:

```
import collections

class Graph:
    def __init__(self):
        self.graph = collections.defaultdict(list)

        global n
        n = int(input("Enter no. of edges in graph: "))

        for i in range(n):
            print(f"Connection {i+1}")
            index = int(input("Enter 1st node: "))
            value = int(input("Enter 2nd node: "))
            print("\n")

            self.graph[index].append(value)

        print(self.graph)

    def BreadthSearch(self, start):
        visited = [0] * n

        queue = []
        print_sequence = []
        queue.append(start)
        visited[start] = 1

        while queue:
            start = queue.pop(0)
            print_sequence.append(start)

            for i in self.graph[start]:
                if visited[i] == 0:
                    queue.append(i)
                    visited[i] = 1
```

```
        for i in print_sequence:
            print(i, end=" ")

g = Graph()
g.BreadthSearch(int(input("Enter starting vertex: ")))
```

Input:

```
Enter no. of edges in graph: 6
Connection 1
Enter 1st node: 0
Enter 2nd node: 1


Connection 2
Enter 1st node: 0
Enter 2nd node: 2


Connection 3
Enter 1st node: 1
Enter 2nd node: 2


Connection 4
Enter 1st node: 2
Enter 2nd node: 0


Connection 5
Enter 1st node: 2
Enter 2nd node: 3


Connection 6
Enter 1st node: 3
Enter 2nd node: 3


defaultdict(<class 'list'>, {0: [1, 2], 1: [2], 2: [0, 3], 3: [3]})
Enter starting vertex: 2
```

Output:

```
2 0 3 1
```

Practical 4

Aim: Write a program to implement Depth First Search algorithm

Define:

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Code:

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v)
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    def DFS(self):
        visited = set()
        for vertex in self.graph:
            if vertex not in visited:
                self.DFSUtil(vertex, visited)

g = Graph()

for i in range(int(input("Enter no. of edges: "))):
    g.addEdge(int(input("Enter 1st node: ")), int(input("Enter 2nd node: ")))
    print("\n")

g.DFS()
```

Input:

```
Enter no. of edges: 6
Enter 1st node: 0
Enter 2nd node: 1


Enter 1st node: 0
Enter 2nd node: 9


Enter 1st node: 1
Enter 2nd node: 2


Enter 1st node: 2
Enter 2nd node: 0


Enter 1st node: 2
Enter 2nd node: 3


Enter 1st node: 9
Enter 2nd node: 3
```

Output:

```

0
1
2
3
9
```

Practical 5

Aim: Write a program to implement Greedy Best First Search

Define:

Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it. For this it uses an evaluation function to decide the traversal.

Code:

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

def best_first_search(source, target, n):
    visited = [0] * n
    pq = PriorityQueue()
    pq.put((0, source))
    while pq.empty() == 0:
        u = pq.get()[1]
        print(u, end=" ")
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == 0:
                visited[v] = 1
                pq.put((c, v))

def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))

addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
```

```
source = 0
target = 9
best_first_search(source, target, v)
```

Input:

```
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
```

Output:

```
0 1 0 3 2 8 9
```

Practical 6

Aim: Write a program to implement A* search algorithm

Define:

A* (pronounced "A-star") is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. It is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

Code:

```
class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    def h(self, n):
        H = {
            'S': 14,
            'B': 12,
            'C': 11,
            'D': 6,
            'E': 4,
            'F': 11,
            'G': 0
        }

        return H[n]

    def a_star_algorithm(self, start, stop):
        open_lst = set([start])
        closed_lst = set([])
        poo = {}
        poo[start] = 0

        par = {}
        par[start] = start
```

```
        while len(open_lst) > 0:
            n = None

            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    n = v

            if n == None:
                print('Path does not exist!')
                return None

            if n == stop:
                reconst_path = []

                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]

                reconst_path.append(start)

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            for (m, weight) in self.get_neighbors(n):
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n
                    poo[m] = poo[n] + weight
                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n

                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)

            open_lst.remove(n)
            closed_lst.add(n)

        print('Path does not exist!')
        return None

adjac_lis = {
    'S': [('B', 4), ('C', 3)],
```

```
    'B': [('F', 5), ('E', 12)],
    'C': [('E', 10), ('D', 7)],
    'D': [('E', 2)],
    'E': [('G', 5)],
    'F': [('G', 16)],
}

graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('S', 'G')
```

Input:

```
adjac_lis = {
    'S': [('B', 4), ('C', 3)],
    'B': [('F', 5), ('E', 12)],
    'C': [('E', 10), ('D', 7)],
    'D': [('E', 2)],
    'E': [('G', 5)],
    'F': [('G', 16)],
}
```

Output:

```
Path found: ['S', 'C', 'D', 'E', 'G']
```

AI Practical 7

Aim: Minimax program with traversing and optimal value as output.

Define:

In the domains of artificial intelligence and game theory, we often come across search problems. Such problems can be described by a graph of interconnected nodes, each representing a possible state. An intelligent agent needs to be able to traverse graphs by evaluating each node to reach a “good” (if not optimal) state. However, there are kinds of problems where typical graph search algorithms cannot be applied. In this tutorial, we’ll discuss such problems and evaluate one of the possible solutions – the Minimax algorithm. The minimax algorithm is a rather simplistic approach to dealing with adversarial problems. “Adversarial” refers to multi-agent systems, where each agent chooses their strategy by considering the likely actions of their opponents. A utility function dictates how “good” a state is for the agent. In a 2-player game, the agent tries to maximize the utility function, while his opponent tries to minimize it. The reasoning behind the algorithm’s name becomes apparent.

Code:

```
import math

def minimax (curDepth, nodeIndex, maxTurn, scores, targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        print(scores[nodeIndex])
        return max(minimax(curDepth + 1, nodeIndex * 2, False, scores,
targetDepth), minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores,
targetDepth))
    else:
        print(scores[nodeIndex])
        return min(minimax(curDepth + 1, nodeIndex * 2, True, scores,
targetDepth), minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores,
targetDepth))

# 3 5 2 9 12 5 23 23
n = int(input("Enter no. of terminal values: "))
scores = []
```

```
print("Enter terminal values: ")
for i in range(n):
    scores.append(int(input(f"Terminal Value {i+1}: ")))
treeDepth = math.log(len(scores), 2)

# print("The optimal value is : \n", end = "")
final = minimax(0, 0, True, scores, treeDepth)
print(f"Final optimal value: {final}")
```

Output:

```
E:\College\SEM VI\AI\AI-Lab>python prac7.py
Enter no. of terminal values: 8
Enter terminal values:
Terminal Value 1: 3
Terminal Value 2: 5
Terminal Value 3: 2
Terminal Value 4: 9
Terminal Value 5: 12
Terminal Value 6: 5
Terminal Value 7: 23
Terminal Value 8: 23
3
3
3
5
5
2
9
Final optimal value: 12
```

AI Practical 8

Aim: N-Queens problem.

Define:

The n-queens puzzle is the problem of placing *n* queens on an *n x n* chessboard such that no two queens attack each other. Given an integer *n*, return all distinct solutions to the n-queens puzzle. You may return the answer in any order.

Code:

```
import copy

def exists(i, j):
    return (i >= 0 and i < n and j >= 0 and j < n)


def contains(i, j, l, m, queen_pairs):
    if ((i, j, l, m) in queen_pairs) or ((l, m, i, j) in queen_pairs):
        return True
    return False

def position_queens_row_wise(board):
    for row in board:
        while row.count(1) > 1:
            for i in range(n):
                if board[i].count(1) == 0:
                    j = row.index(1)
                    board[i][j] = 1
                    row[j] = 0
                    break

    return board


def heuristic_value(board):
    h = 0
    queen_pairs = []
    for i in range(n):
        for j in range(n):

            if board[i][j]:
                for k in range(n):
                    if board[i][k] == 1 and k != j and not contains(i, j, i,
k, queen_pairs):
                        queen_pairs.append((i, j, i, k))
```

```
                        h += 1

                for k in range(n):
                    if board[k][j] == 1 and i != k and not contains(i, j, k,
j, queen_pairs):
                        queen_pairs.append((i, j, k, j))
                        h += 1

                l, m = i-1, j+1
                while exists(l, m):
                    if board[l][m] == 1 and not contains(i, j, l, m,
queen_pairs):
                        queen_pairs.append((i, j, l, m))
                        h += 1
                    l, m = l-1, m+1

                l, m = i+1, j-1
                while exists(l, m):
                    if board[l][m] == 1 and not contains(i, j, l, m,
queen_pairs):
                        queen_pairs.append((i, j, l, m))
                        h += 1
                    l, m = l+1, m-1

                l, m = i-1, j-1
                while exists(l, m):
                    if board[l][m] == 1 and not contains(i, j, l, m,
queen_pairs):
                        queen_pairs.append((i, j, l, m))
                        h += 1
                    l, m = l-1, m-1

                l, m = i+1, j+1
                while exists(l, m):
                    if board[l][m] == 1 and not contains(i, j, l, m,
queen_pairs):
                        queen_pairs.append((i, j, l, m))
                        h += 1
                    l, m = l+1, m+1

    return h


def hill_climbing(board):
    min_board = board
    min_h = 999999
    global n_side_moves, n_steps

    n_steps += 1
```

```
    if n_side_moves == 100:
        return -1

    sideway_move = False

    for i in range(n):
        queen = board[i].index(1)

        board[i][queen] = 0

        for k in range(n):

            if k != queen:
                board[i][k] = 1

                h = heuristic_value(board)

                if h < min_h:
                    min_h = h
                    min_board = copy.deepcopy(board)
                if h == min_h:
                    min_h = h
                    min_board = copy.deepcopy(board)
                    sideway_move = True

                board[i][k] = 0

        board[i][queen] = 1

    if sideway_move:
        n_side_moves += 1

    if min_h == 0:
        print("Number of steps required: {}".format(n_steps))
        return min_board

    return hill_climbing(min_board)

n_side_moves = 0
n_steps = 0

n = int(input("Enter no. of sides:"))
board = [
    [1] * n
]

for i in range(n-1):
```

```
    board.append([0] * n)

print("Starting Board:\n")
for i in board:
    print(f"{i}\n")

board = position_queens_row_wise(board)
board = hill_climbing(board)

print("Final Board:\n")
for i in board:
    print(f"{i}\n")
```

Output:

```
E:\College\SEM VI\AI\AI-Lab>python prac8.py
Enter no. of sides:8
Starting Board:

[1, 1, 1, 1, 1, 1, 1, 1]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 0]

Number of steps required: 11
Final Board:

[0, 0, 1, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 1, 0, 0]

[0, 1, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 1, 0]

[1, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 1]

[0, 0, 0, 0, 1, 0, 0, 0]
```

AI Practical 9

Aim: Write Water jug problem in SWI-Prolog.

Define:

Water pouring puzzles (also called water jug problems, decanting problems, measuring puzzles, or Die Hard with a Vengeance puzzles) are a class of puzzle involving a finite collection of water jugs of known integer capacities (in terms of a liquid measure such as litres or gallons). Initially each jug contains a known integer volume of liquid, not necessarily equal to its capacity. Puzzles of this type ask how many steps of pouring water from one jug to another (until either one jug becomes empty or the other becomes full) are needed to reach a goal state, specified in terms of the volume of liquid that must be present in some jug or jugs. By Bézout's identity, such puzzles have solution if and only if the desired volume is a multiple of the greatest common divisor of all the integer volume capacities of jugs.

Code:

```
reverse([],[]).
reverse([H|T], ReversedList):-
    reverse(T, RevTemp), append(RevTemp, [H], ReversedList).

display_list([]).
display_list([A|B]) :-
  write(A),
  display_list(B).

state(2,_,_,Moves):-
        reverse(Moves, Rev),
        display_list(Rev),
        !.
state(X,Y, Visited, Moves):- X < 4,
        not(member([4,Y],Visited)),
        state(4,Y,[[4,Y]|Visited], ["Fill the 4-Gallon Jug\n"|Moves]).


state(X,Y, Visited, Moves):- Y < 3,
        not(member([X,3],Visited)),
        state(X,3,[[X,3]|Visited], ["Fill the 3-Gallon Jug\n"|Moves]).

state(X,Y, Visited, Moves):- X > 0,
        not(member([0,Y],Visited)),
        state(0,Y, [[0,Y]|Visited], ["Empty the 4-Gallon jug on
ground\n"|Moves] ).
```

```
state(X,Y, Visited, Moves):- Y > 0,
        not(member([X,0],Visited)),
        state(X,0,[[X,0]|Visited], ["Empty the 3-Gallon jug on
ground\n"|Moves] ).

state(X,Y, Visited, Moves):- X + Y >= 4,
        Y > 0,
        NEW_Y is Y - (4 - X),
        not(member([4,NEW_Y],Visited)),
        state(4,NEW_Y,[[4,NEW_Y]|Visited], ["Pour water from 3-Gallon jug to
4-gallon until it is full\n"|Moves] ).

state(X,Y, Visited, Moves):- X + Y >=3,
        X > 0,
        NEW_X is X - (3 - Y),
        not(member([NEW_X,3],Visited)),
        state(NEW_X,3,[[NEW_X,3]|Visited], ["Pour water from 4-Gallon jug to
3-gallon until it is full\n"|Moves] ).

state(X,Y, Visited, Moves):- X + Y =< 4,
        Y > 0,
        NEW_X is X + Y,
        not(member([NEW_X,0],Visited)),
        state(NEW_X,0,[[NEW_X,0]|Visited] , ["Pour all the water from 3-Gallon
jug to 4-gallon\n"|Moves] ).

state(X,Y, Visited, Moves):- X + Y =< 3,
        X > 0,
        NEW_Y is X + Y,
        not(member([0,NEW_Y],Visited)),
        state(0,NEW_Y, [[0,NEW_Y]|Visited], ["Pour all the water from 4-Gallon
jug to 3-gallon\n"|Moves]  ).

state(0,2, Visited, Moves):-
        not(member([2,0],Visited)),
        state(2,0,[[2,0]|Visited], ["Pour 2 gallons from 3-Gallon jug to 4-
gallon\n"|Moves] ).

state(2,Y,Visited, Moves):-
        not(member([0,Y],Visited)),
        state(0,Y,[[0,Y]|Visited], ["Empty 2 gallons from 4-Gallon jug on the
ground\n"|Moves] ).
```

Output:

```
?- jugs(4, 3).
Empty the 4-Gallon jug on ground
Empty the 3-Gallon jug on ground
Fill the 4-Gallon Jug
Pour water from 4-Gallon jug to 3-gallon until it is full
Empty the 3-Gallon jug on ground
Pour all the water from 4-Gallon jug to 3-gallon
Fill the 4-Gallon Jug
Pour water from 4-Gallon jug to 3-gallon until it is full
true .
```

AI Practical 10

Aim: Write a Problem for Monkey Banana problem.

Define:

The monkey and banana problem are a famous toy problem in artificial intelligence, particularly in logic programming and planning. A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey? The problem seeks to answer the question of whether monkeys are intelligent. Both humans and monkeys have the ability to use mental maps to remember things like where to go to find shelter, or how to avoid danger. They can also remember where to go to gather food and water, as well as how to communicate with each other. Monkeys have the ability not only to remember how to hunt and gather but to learn new things, as is the case with the monkey and the bananas: despite the fact that the monkey may never have been in an identical situation, with the same artifacts at hand, a monkey is capable of concluding that it needs to make a ladder, position it below the bananas, and climb up to reach for them.

Code:

```
do( state(middle, onbox, middle, hasnot),   % grab banana
    grab,
    state(middle, onbox, middle, has) ).

do( state(L, onfloor, L, Banana),           % climb box
    climb,
    state(L, onbox, L, Banana) ).

do( state(L1, onfloor, L1, Banana),         % push box from L1 to L2
    push(L1, L2),
    state(L2, onfloor, L2, Banana) ).

do( state(L1, onfloor, Box, Banana),        % walk from L1 to L2
    walk(L1, L2),
    state(L2, onfloor, Box, Banana) ).


% canget(State): monkey can get banana in State
```

```
canget(state(_, _, _, has)).                % Monkey already has it, goal
state

canget(State1) :-                           % not goal state, do some work to
get it
      do(State1, Action, State2),           % do something (grab, climb, push,
walk)

% get plan = list of actions

canget(state(_, _, _, has), []).            % Monkey already has it, goal
state

canget(State1, Plan) :-                     % not goal state, do some work to
get it
      do(State1, Action, State2),           % do something (grab, climb, push,
walk)
      canget(State2, PartialPlan),          % canget from State2
      add(Action, PartialPlan, Plan).       % add action to Plan

add(X,L,[X|L]).
```

Output:

```
?- canget(state(atwindow, onbox, atwindow, hasnot), Plan).
false.
```

AI Practical 11

Aim: Write a problem for implementing Expert system to diagnose medical disease.

Define:

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. It solves the most complex issue as an expert by extracting the knowledge stored in its knowledge base. The system helps in decision making for complex problems using both facts and heuristics like a human expert. It is called so because it contains the expert knowledge of a specific domain and can solve any complex problem of that domain. These systems are designed for a specific domain, such as medicine, science, etc. The performance of an expert system is based on the expert's knowledge stored in its knowledge base. The more knowledge stored in the KB, the more that system improves its performance. One of the common examples of an ES is a suggestion of spelling errors while typing in the Google search box.

Code:

```
go :-
write('What is the patient''s name? '),
read(Patient),get_single_char(Code),
hypothesis(Patient,Disease),
write_list([Patient,', probably has ',Disease,'.']),nl.

go :-
write('Sorry, I don''t seem to be able to'),nl,
write('diagnose the disease.'),nl.

symptom(Patient,fever) :-
verify(Patient," have a fever (y/n) ?").
symptom(Patient,rash) :-
verify(Patient," have a rash (y/n) ?").
symptom(Patient,headache) :-
verify(Patient," have a headache (y/n) ?").
symptom(Patient,runny_nose) :-
verify(Patient," have a runny_nose (y/n) ?").
symptom(Patient,conjunctivitis) :-
verify(Patient," have a conjunctivitis (y/n) ?").
symptom(Patient,cough) :-
```

```
verify(Patient," have a cough (y/n) ?").
symptom(Patient,body_ache) :-
verify(Patient," have a body_ache (y/n) ?").
symptom(Patient,chills) :-
verify(Patient," have a chills (y/n) ?").
symptom(Patient,sore_throat) :-
verify(Patient," have a sore_throat (y/n) ?").
symptom(Patient,sneezing) :-
verify(Patient," have a sneezing (y/n) ?").
symptom(Patient,swollen_glands) :-
verify(Patient," have a swollen_glands (y/n) ?").

ask(Patient,Question) :-
      write(Patient),write(', do you'),write(Question),
      read(N),
      ( (N == yes ; N == y)
      ->
       assert(yes(Question)) ;
       assert(no(Question)), fail).

:- dynamic yes/1,no/1.

verify(P,S) :-
   (yes(S) -> true ;
    (no(S) -> fail ;
     ask(P,S))).

undo :- retract(yes(_)),fail.
undo :- retract(no(_)),fail.
undo.



hypothesis(Patient,german_measles) :-
symptom(Patient,fever),
symptom(Patient,headache),
symptom(Patient,runny_nose),
symptom(Patient,rash).

hypothesis(Patient,common_cold) :-
symptom(Patient,headache),
symptom(Patient,sneezing),
symptom(Patient,sore_throat),
symptom(Patient,runny_nose),
symptom(Patient,chills).

hypothesis(Patient,measles) :-
symptom(Patient,cough),
symptom(Patient,sneezing),
```

```
symptom(Patient,runny_nose).

hypothesis(Patient,flu) :-
symptom(Patient,fever),
symptom(Patient,headache),
symptom(Patient,body_ache),
symptom(Patient,conjunctivitis),
symptom(Patient,chills),
symptom(Patient,sore_throat),
symptom(Patient,runny_nose),
symptom(Patient,cough).



hypothesis(Patient,mumps) :-
symptom(Patient,fever),
symptom(Patient,swollen_glands).

hypothesis(Patient,chicken_pox) :-
symptom(Patient,fever),
symptom(Patient,chills),
symptom(Patient,body_ache),
symptom(Patient,rash).

write_list([]).
write_list([Term| Terms]) :-
write(Term),
write_list(Terms).

response(Reply) :-
get_single_char(Code),
put_code(Code), nl,
char_code(Reply, Code).
```

Output:

```
?- go.
What is the patient's name? Kartik
|: .
_5908, do you have a fever (y/n) ?n
|: .
_5908, do you have a headache (y/n) ?|: n.
_5908, do you have a cough (y/n) ?|: y.
_5908, do you have a sneezing (y/n) ?|: y.
_5908, do you have a runny_nose (y/n) ?|: n.
Sorry, I don't seem to be able to
diagnose the disease.
true.
```