Introduction to Prolog

What is Prolog

Prolog stands for programming in logic. In the logic programming paradigm, prolog language is most widely available. Prolog is a declarative language, which means that a program consists of data based on the facts and rules (Logical relationship) rather than computing how to find a solution. A logical relationship describes the relationships which hold for the given application. To obtain the solution, the user asks a question rather than running a program. When a user asks a question, then to determine the answer, the run time system searches through the database of facts and rules.

The first Prolog was '**Marseille Prolog**', which is based on work by Colmerauer. The major example of fourth-generation programming language was prolog. It supports the declarative programming paradigm.

In 1981, a Japanese computer Project of 5th generation was announced. After that, it was adopted Prolog as a development language. In this tutorial, the program was written in the 'Standard' Edinburgh Prolog. Prologs of PrologII family are the other kind of prologs which are descendants of Marseille Prolog.

Prolog features are 'Logical variable', which means that they behave like uniform data structure, a backtracking strategy to search for proofs, a pattern-matching facility, mathematical variable, and input and out are interchangeable.

To deduce the answer, there will be more than one way. In such case, the run time system will be asked to find another solution. To generate another solution, use the backtracking strategy. Prolog is a weakly typed language with static scope rules and dynamic type checking.

Prolog is a declarative language that means we can specify what problem we want to solve rather than how to solve it.

Prolog is used in some areas like database, natural language processing, artificial intelligence, but it is useless in some areas like a numerical algorithm or instance graphics.

In artificial intelligence applications, prolog is used. The artificial intelligence applications can be automated reasoning systems, natural language interfaces, and expert systems. The expert system consists of an interface engine and a database of facts. The prolog's run time system provides the service of an interface engine.
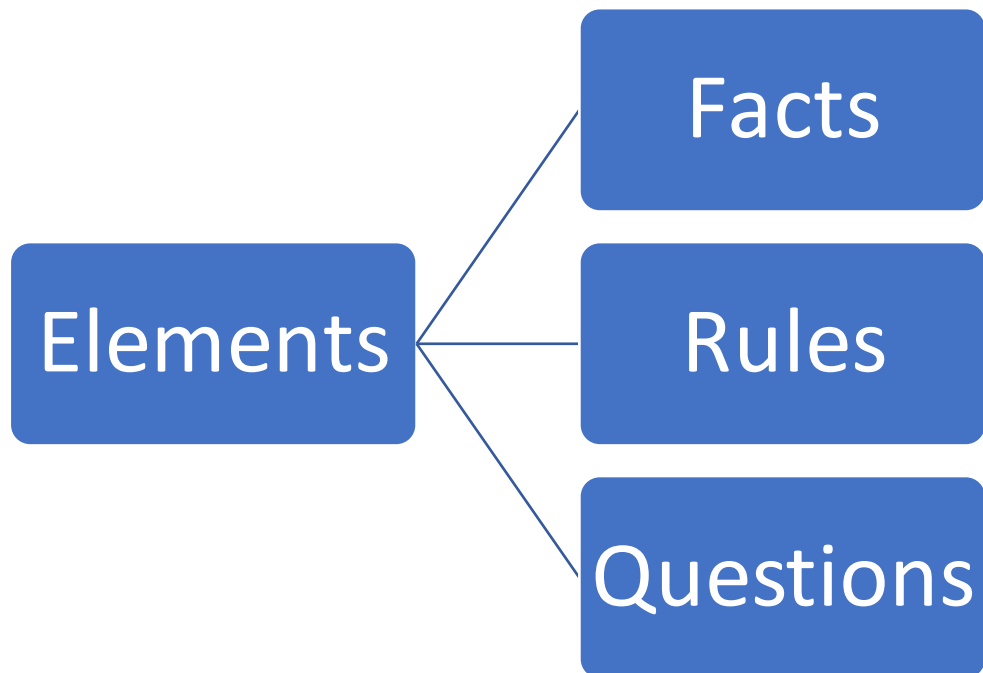
A basic logic programming environment has no literal values. An identifier with upper case letters and other identifiers denote variables. Identifiers that start with lower-case letters denote data values. The basic Prolog elements are typeless. The most implementations of prolog have been enhanced to include integer value, characters, and operations. The Mechanism of prolog describes the tuples and lists.

Functional programming language and prolog have some similarities like Hugs. A logic program is used to consist of relation definition. A functional programming language is used to consist of a sequence of function definitions. Both the logical programming and functional programming rely heavily on recursive definitions.

Prolog or PROgramming in LOGics is a logical and declarative programming language. It is one major example of the fourth-generation language that supports the declarative programming paradigm. This is particularly suitable for programs that involve symbolic or non-numeric computation. This is the main reason to use Prolog as the programming language in Artificial Intelligence, where symbol manipulation and inference manipulation are the fundamental tasks.

In Prolog, we need not mention the way how one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it. However, in Prolog we are supposed to give clues as the solution method.

Prolog language basically has three different elements –

```
                          ┌──────────┐
                          │  Facts   │
                          └──────────┘
┌──────────┐              ┌──────────┐
│ Elements │──────────────│  Rules   │
└──────────┘              └──────────┘
                          ┌───────────┐
                          │ Questions │
                          └───────────┘
```

Facts
The fact is predicate that is true, for example, if we say, "Tom is the son of Jack", then this is a fact.

Rules
Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as –

grandfather (X, Y) :- father(X, Z), parent(Z, Y)

This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be father of Z.

Questions
And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

Applications of Prolog

Prolog is used in various domains. It plays a vital role in automation system. Following are some other important fields where Prolog is used −

- Intelligent Database Retrieval

- Natural Language Understanding

- Specification Language

- Machine Learning

- Robot Planning

- Automation System

- Problem Solving

Getting Started

SWI-Prolog is installed as 'swipl'. SWI-Prolog is normally operated as an interactive application simply by starting the program:

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
```

- **Defining rules**

In SWI Prolog we can define rules and facts according to our convenience of whatever we want.

For example:

```prolog
% Fathers Side
male(varun).
male(karan).
male(yogesh).
male(mahendra).
male(prashant).
male(sanjay).
male(yash).
male(arvind).

% Fathers Side
female(amita).
female(jyoti).
female(deepal).
female(taruni).
female(archana).
female(arpita).
female(hemlata).

% Fathers Side
parent_of(mahendra,varun).
parent_of(amita,varun).
parent_of(jyoti,karan).
parent_of(jyoti,deepal).
parent_of(yogesh,karan).
parent_of(yogesh,deepal).
parent_of(prashant,yash).
parent_of(prashant,arpita).
parent_of(taruni,yash).
parent_of(taruni,arpita).
parent_of(arvind,mahendra).
parent_of(arvind,yogesh).
parent_of(arvind,prashant).
parent_of(arvind,sanjay).
parent_of(hemlata,yogesh).
parent_of(hemlata,mahendra).
parent_of(hemlata,sanjay).
parent_of(hemlata,prashant).

% Fathers Side
married(yogesh,jyoti).
married(mahendra,amita).
married(prashant,taruni).
married(sanjay,archana).
married(arvind,hemlata).
```

```
%self roles rule
father(X, Y) :- male(X), parent_of(X, Y).
mother(X,Y) :- female(X), parent_of(X,Y).

brother(X, Y) :- male(X), father(Z, Y), father(Z, X), X \= Y.
brother(X, Y) :- male(X), mother(Z, Y), mother(Z, X), X \= Y.

sister(X, Y) :- female(X), father(Z, Y), father(Z, X), X \= Y.
sister(X, Y) :- female(X), mother(Z, Y), mother(Z, X), X \= Y.

uncle_of(X,Y) :- parent_of(Z,Y), brother(Z,X).

grandparent(X, Y) :- parent_of(X, Z), parent_of(Z, Y).

grandmother(X, Y) :- parent_of(X, Z), parent_of(Z, Y),female(X).
grandfather(X, Y) :- parent_of(X, Z), parent_of(Z, Y),male(X).

aunt_of(X,Y) :- female(X), father(Z,Y), brother(Z,W), married(W,X).
```

And to verify any code using the codes then:

```
?- parent_of(X,varun).
X = mahendra ;
X = amita.

?- uncle_of(X,deepal).
X = mahendra ;
X = prashant ;
X = sanjay ;
X = mahendra ;
X = sanjay ;
X = prashant.

?- grandparent(X,varun).
X = arvind ;
X = hemlata .

?- aunt_of(X,karan).
X = amita ;
X = amita ;
X = taruni ;
X = taruni ;
X = archana ;
X = archana ;
false.

?-
```

Write a program to implement Family Tree

Code

```
% FATHERS SIDE
MALE(VARUN).
MALE(KARAN).
MALE(YOGESH).
MALE(MAHENDRA).
MALE(PRASHANT).
MALE(SANJAY).
MALE(YASH).
MALE(ARVIND).

FEMALE(AMITA).
FEMALE(JYOTI).
FEMALE(DEEPAL).
FEMALE(TARUNI).
FEMALE(ARCHANA).
FEMALE(ARPITA).
FEMALE(HEMLATA).

PARENT_OF(MAHENDRA,VARUN).
PARENT_OF(AMITA,VARUN).
PARENT_OF(JYOTI,KARAN).
PARENT_OF(JYOTI,DEEPAL).
PARENT_OF(YOGESH,KARAN).
PARENT_OF(YOGESH,DEEPAL).
PARENT_OF(PRASHANT,YASH).
PARENT_OF(PRASHANT,ARPITA).
PARENT_OF(TARUNI,YASH).
PARENT_OF(TARUNI,ARPITA).
PARENT_OF(ARVIND,MAHENDRA).
PARENT_OF(ARVIND,YOGESH).
PARENT_OF(ARVIND,PRASHANT).
PARENT_OF(ARVIND,SANJAY).
PARENT_OF(HEMLATA,YOGESH).
PARENT_OF(HEMLATA,MAHENDRA).

MARRIED(YOGESH,JYOTI).
MARRIED(MAHENDRA,AMITA).
MARRIED(PRASHANT,TARUNI).
MARRIED(SANJAY,ARCHANA).
MARRIED(ARVIND,HEMLATA).
```

%SELF ROLES RULE
FATHER(X, Y) :- MALE(X), PARENT_OF(X, Y).
MOTHER(X,Y) :- FEMALE(X), PARENT_OF(X,Y).

BROTHER(X, Y) :- MALE(X), FATHER(Z, Y), FATHER(Z, X), X \= Y.
BROTHER(X, Y) :- MALE(X), MOTHER(Z, Y), MOTHER(Z, X), X \= Y.

SISTER(X, Y) :- FEMALE(X), FATHER(Z, Y), FATHER(Z, X), X \= Y.
SISTER(X, Y) :- FEMALE(X), MOTHER(Z, Y), MOTHER(Z, X), X \= Y.

UNCLE_OF(X,Y) :- PARENT_OF(Z,Y), BROTHER(Z,X).

GRANDPARENT(X, Y) :- PARENT_OF(X, Z), PARENT_OF(Z, Y).

GRANDMOTHER(X, Y) :- PARENT_OF(X, Z), PARENT_OF(Z, Y),FEMALE(X).
GRANDFATHER(X, Y) :- PARENT_OF(X, Z), PARENT_OF(Z, Y),MALE(X).

AUNT_OF(X,Y) :- FEMALE(X), FATHER(Z,Y), BROTHER(Z,W), MARRIED(W,X).

OUTPUT

```
SWI-Prolog (AMD64, Multi-threaded, version 8.4.1)

File  Edit  Settings  Run  Debug  Help
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/users/varun/documents/prolog/prac_2_family_tree compiled 0.00 sec, -2 clauses
?-
|    father(X,varun).
X = mahendra ,

?- mother(X,varun).
X = amita ,

?- aunt_of(X,varun).
X = jyoti ;
X = jyoti ;
X = taruni ;
X = archana ;
false.

?- grandparent(X,varun).
X = arvind ;
X = hemlata ,

?- grandfather(X,varun).
X = arvind ,

?- uncle_of(X,varun).
X = yogesh ;
X = prashant ;
X = sanjay ,
```

To implement Breadth First Search

Theory

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

- First move horizontally and visit all the nodes of the current layer
- Move to the next layer
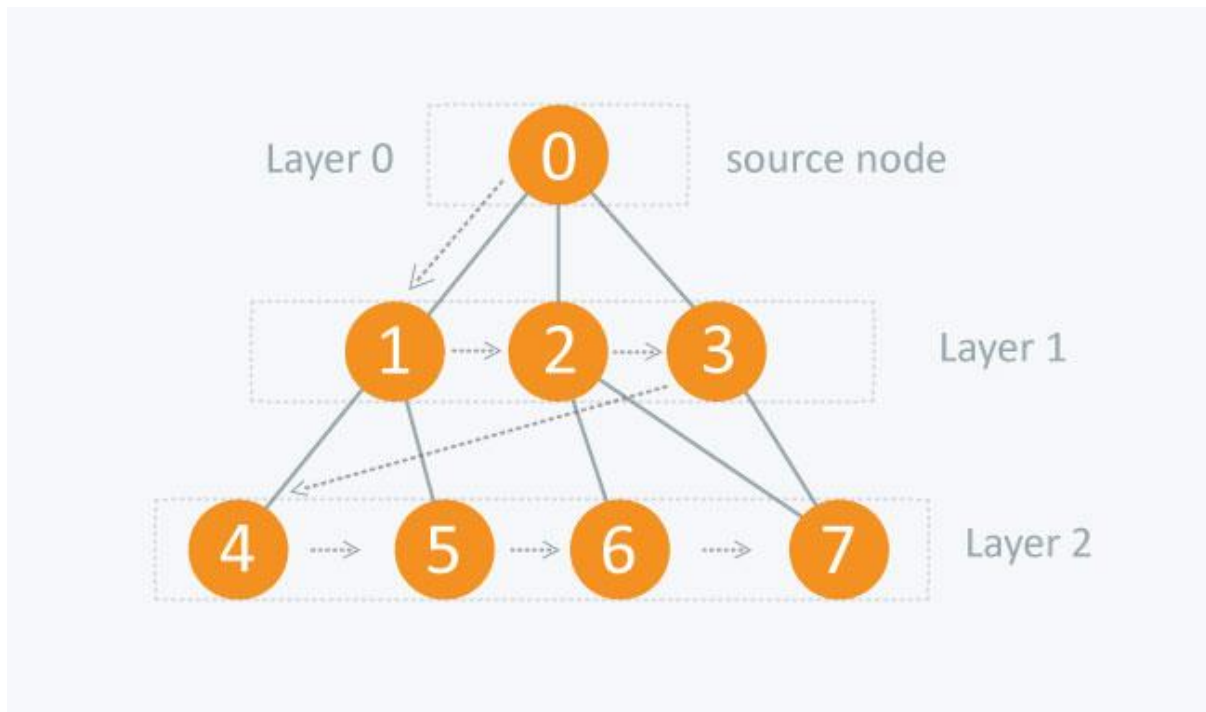
The distance between the nodes in layer 1 is comparitively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neigbors of the node will be

visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Complexity

The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of edges.

Code

```c
#include<stdio.h>
void adj();
void input();
void bfs(int);
int delQue();
void addQue(int);
int v,n,f=0,r=0,visited[10]={0},a[10][10],que[10]={0};
int main()
{
   printf("\tBFS TRAVERSAL\n\n");
       input();
       printf("\nEnter the starting vertex: \n");
       scanf("%d",&v);
       printf("\nBFS traversal Path: \n");
       bfs(v);
       return 0;
}
void input()
{
       int i, j;
       printf("\nEnter the number of nodes: ");
       scanf("%d",&n);
       printf("\nEnter the adjacent matrix: \n");
       for(i=1;i<=n;i++)
               for(j=1;j<=n;j++)
                       scanf("%d",&a[i][j]);
}
```

```c
void bfs(int v)
{
    printf("%d->", v);
    int j,u=v;
    visited[u]=1;
    while(1)
    {
        for(j=1; j<=n; j++)
        {
            if(a[u][j]!=0 && visited[j]==0)
            {
                visited[j]=1;
                addQue(j);
            }
        }
        u=delQue();
        printf("%d->", u);
        if(f==r)
            break;
    }
    printf("NULL\n");
}

void adj()
{
    int i, j;
    printf("\nAdjacency Matrix[][]: \n");
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
            printf("%d\t", a[i][j]);
        printf("\n");
```

```
        }
}


void addQue(int x)
{
        que[r++]=x;
}


int delQue()
{
        return que[f++];
}
```

Output
Taking this example
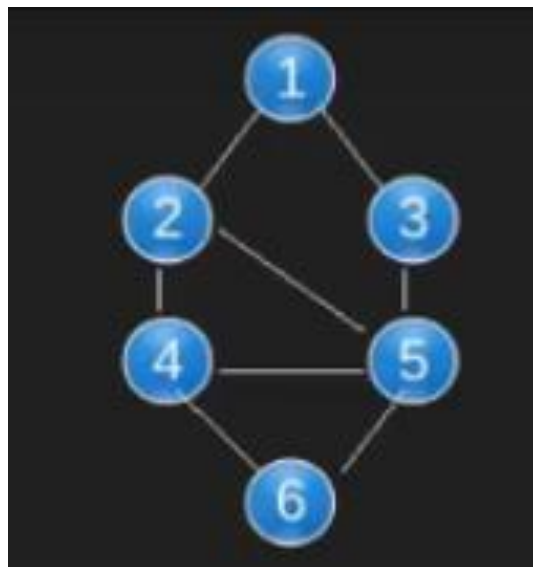
```
        BFS TRAVERSAL


Enter the number of nodes: 6

Enter the adjacent matrix:
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 0 1 0
0 1 0 0 1 1
0 1 0 1 0 1
0 0 0 1 1 0

Enter the starting vertex:
1

BFS traversal Path:
1->2->3->4->5->6->NULL

Process returned 0 (0x0)   execution time : 110.237 s
Press any key to continue.
```

To implement Depth First Search

Theory

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks.

The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- Repeat this process until the stack is empty.

However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Complexity

Time complexity O(V+E), when implemented using an adjacency list.

Code

```
#include <stdio.h>
#include <stdlib.h>


int n; // number of nodes in graph
int *visited; // array to keep a track of nodes visited


void dfs(int i, int g[n][n]);


int main()
{
    int i, j, node;


    printf("Enter the number of nodes: ");
    scanf("%d", &n);
```

```c
        int g[n][n];
        visited = (int *)calloc(n, sizeof(int));


        printf("Enter the adjacency matrix of the graph:\n");
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                        scanf("%d", &g[i][j]);
        }


        printf("Enter the source node: ");
        scanf("%d", &node);
        printf("\nThe nodes reachable from %d are:\n", node);
        dfs(node-1, g);


        return 0;
}


void dfs(int i, int g[n][n])
{
        int j;


        visited[i] = 1;
        printf("%d ", i+1);


        for (j = 0; j < n; j++)
        {
                if (!visited[j] && g[i][j] == 1)
                        dfs(j,g);
        }
}
```
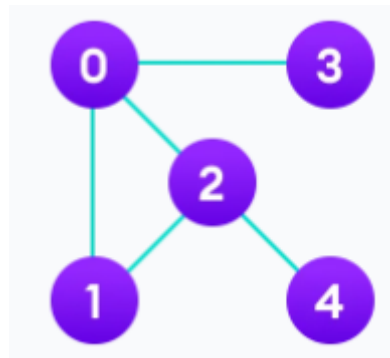
Taking this example

Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 10-1-2021	Date of Submission: 31-1-2022

To implement Greedy Best First Search

Theory

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

$$f(n)= g(n).$$

Were, h(n)= estimated cost from node n to the goal.
The greedy best first algorithm is implemented by the priority queue.

Steps to follow

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Code

```python
from queue import PriorityQueue
v = 5
graph = [[] for i in range(v)]
def best_first_search(source, target, n):
  visited = [0] * n
  visited[0] = True
  pq = PriorityQueue()
  pq.put((0, source))
  while pq.empty() == False:
    u = pq.get()[1]
    print(u, end=" ")
    if u == target:
      break
    for v, c in graph[u]:
      if visited[v] == False:
        visited[v] = True
        pq.put((c, v))
  print()
```

```python
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
addedge(0, 1, 5)
addedge(0, 2, 1)
addedge(2, 3, 2)
addedge(1, 4, 1)
addedge(3, 4, 2)
source = 0
target = 4
best_first_search(source, target, v)
```

Output

0 2 3 4

To implement A* Algorithm

Theory

This Algorithm is the advanced form of the BFS algorithm (Breadth-first search), which searches for the shorter path first than, the longer paths. It is a complete as well as an optimal solution for solving path and grid problems.

$$f(n) = g(n) + h(n)$$

Where

g (n): The actual cost path from the start node to the current node.

h (n): The actual cost path from the current node to goal node.

f (n): The actual cost path from the start node to the goal node.

Algorithm

1. Firstly, Place the starting node into OPEN and find its f (n) value.
2. Then remove the node from OPEN, having the smallest f (n) value. If it is a goal node, then stop and return to success.
3. Else remove the node from OPEN and find all its successors.
4. Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.
5. Goto Step-2.
6. Exit.

Code

```python
from collections import deque

class Graph:

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

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

    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]
```

```python
def a_star_algorithm(self, start_node, stop_node):

    open_list = set([start_node])
    closed_list = set([])

    g = {}

    g[start_node] = 0

    parents = {}
    parents[start_node] = start_node

    while len(open_list) > 0:
        n = None

        for v in open_list:
            if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                n = v;

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


        if n == stop_node:
            reconst_path = []

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

            reconst_path.append(start_node)

            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_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight


            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
```

```python
                        parents[m] = n

                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)

            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

Output

```
PS E:\TY\SEM VI\AI> & C:/Users/varun/AppData/Local/Programs/Python/Python310/python.exe "e:/TY/SEM VI/AI/PRAC_4.py"
Path found: ['A', 'B', 'D']
```

Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 21-2-2021	Date of Submission: 21-2-2022

To implement MiniMax Algorithm

○ Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.

○ Mini-Max algorithm uses recursion to search through the game-tree.

○ Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.

○ In this algorithm two players play the game, one is called MAX and other is called MIN.

○ Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.

○ Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.

○ The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.

○ The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Algorithm

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
maxEva= -infinity
 for each child of node do
 eva= minimax(child, depth-1, false)
maxEva= max(maxEva,eva)      //gives Maximum of the values
return maxEva

else                // for Minimizer player
```

```
minEva= +infinity
for each child of node do
eva= minimax(child, depth-1, true)
minEva= min(minEva, eva)       //gives minimum of the values
return minEva
```

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Code

```python
import math
def minimax (curDepth, nodeIndex,maxTurn, scores,targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]
    if (maxTurn):
        print (scores[nodeIndex])
        print ("->")
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))
    else:
        print (scores[nodeIndex])
        print (" ->")
        return min(minimax(curDepth + 1, nodeIndex * 2,
                    True, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    True, scores, targetDepth))

scores = [2, 3, 5, 9, 0, 1, 7, 5]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")
result=minimax(0, 0, True, scores, treeDepth)
print(result)
```

Output

```
PS E:\TY\SEM VI\AI> & C:/Users/varun/AppData/Local/Programs/Python/Python310/python.exe "e:/TY/SEM VI/AI/minimaxalgo.py"
The optimal value is : 3
 ->
3
 ->
3
 ->
5
 ->
5
 ->
2
 ->
9
 ->
12
->
3
->
3
 ->
5
->
9
->
3
```

To implement N Queens Problem

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.
The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following are two solutions for 4 Queen problem.

Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.
1) Start in the leftmost column
2) If all queens are placed return true
3) Try all rows in the current column. Do following for every tried row.
   a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   b) If placing queen in [row, column] leads to a solution then return true.
   c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
4) If all rows have been tried and nothing worked, return false to trigger backtracking.

Code

```
#include <stdio.h>
#include <stdlib.h>

int NoSoln(int k, int col[])
{
   int i;
   for(i=1;i<=k-1;i++)
   {
     if(col[k]==col[i] || (abs(i-k)==abs(col[i] - col[k])))
        return 1;
   }
   return 0;
}


int NQueen(int n)
{
   int k = 1;
   int count=0;
   int i,j,col[n+1];
   col[k]=0;
   while(k!=0)
```

```c
{
    col[k] += 1;
    while(col[k]<=n && NoSoln(k,col))
        col[k]=col[k]+1;
    if(col[k]<=n)
    {
        if(k==n)
        {
            count++;
            printf("\nSolution - %d : \n",count);
            for(i=1;i<=n;i++)
            {
                for(j=1;j<=n;j++)
                    if(col[i] == j)
                        printf(" Q%d",i);
                    else
                        printf(" * ");
                printf("\n\n");
            }
        }
        else
        {
            k++;
            col[k]=0;
        }
    }
    else
        k--;
    }
    return count;
}

int main()
{
    int n,solutions;
    printf("\tN-Queens Problem");
    printf("\nEnter the number of queens : ");
    scanf("%d",&n);
    solutions=NQueen(n);
    if(solutions==0)
        printf("No solution!!");
    return 0;
}
```

Output

```
          N-Queens Problem
Enter the number of queens : 4

Solution - 1 :
 *   Q1  *   *

 *   *   *   Q2

 Q3  *   *   *

 *   *   Q4  *


Solution - 2 :
 *   *   Q1  *

 Q2  *   *   *

 *   *   *   Q3

 *   Q4  *   *
```

Write a program for water jug problem using Pruning Technique

Water Jug Problem is also known as Water Pouring Puzzles, measuring puzzles and decanting problems. These belong to a class of puzzles, in which there are a finite and specific number of water jugs having predefined integral capacities, in terms of gallons or litres.

The prime challenge in the water jug problem is that these water jugs do not have any calibrations for measuring the intermediate water levels. In order to solve these puzzles, you are given a required measurement that you have to achieve by transferring waters from one jug to another, you can iterate multiple times until the final goal is reached, but in order to get the best results, the final solution should have a minimum cost and a minimum number of transfers.

Problem Statement:

Given two water jugs with capacities **X** and **Y** liters. Initially, both the jugs are empty. Also given that there is an infinite amount of water available. The jugs do not have markings to measure smaller quantities.

One can perform the following operations on the jug:

- Fill any of the jugs completely with water.
- Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X – d, Y + d)
- Empty any of the jugs

The task is to determine whether it is possible to measure **Z** liters of water using both the jugs. And if true, print any of the possible ways.

Algorithm

- Initialise a queue to implement BFS.
- Since, initially, both the jugs are empty, insert the state {0, 0} into the queue.
- Perform the following state, till the queue becomes empty:
    - Pop out the first element of the queue.
    - If the value of popped element is equal to Z, return True.
    - Let X_left and Y_left be the amount of water left in the jugs respectively.
    - Now perform the fill operation:
        - If the value of X_left < X, insert (({X_left, Y}) into the hashmap, since this state hasn't been visited and some water can still be poured in the jug.
        - If the value of Y_left < Y, insert (({Y_left, X}) into the hashmap, since this state hasn't been visited and some water can still be poured in the jug.
    - Perform the **empty** operation:
        - If the state (({0, Y_left}) isn't visited, insert it into the hashmap, since we can empty any of the jugs.

- Similarly, if the state ({X_left, 0) isn't visited, insert it into the hashmap, since we can empty any of the jugs.
  - Perform the **transfer of water** operation:
    - min({X-X_left, Y}) can be poured from second jug to first jug. Therefore, in case – {X + min({X-X_left, Y}) , Y – min({X-X_left, Y}) isn't visited, put it into hashmap.
    - min({X_left, Y-Y_left}) can be poured from first jug to second jug. Therefore, in case – {X_left – min({X_left, Y – X_left}) , Y + min({X_left, Y – Y_left}) isn't visited, put it into hashmap.
- Return False, since, it is not possible to measure **Z** litres.

Code

```python
capacity = (12,8,5)
x = capacity[0]
y = capacity[1]
z = capacity[2]

memory = {}

ans = []

def get_all_states(state):
    a = state[0]
    b = state[1]
    c = state[2]

    if(a==6 and b==6):
        ans.append(state)
        return True

    if((a,b,c) in memory):
        return False

    memory[(a,b,c)] = 1

    if(a>0):
        if(a+b<=y):
            if( get_all_states((0,a+b,c)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((a-(y-b), y, c)) ):
                ans.append(state)
                return True
        if(a+c<=z):
            if( get_all_states((0,b,a+c)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((a-(z-c), b, z)) ):
```

```python
                    ans.append(state)
                    return True

    if(b>0):
        if(a+b<=x):
            if( get_all_states((a+b, 0, c)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((x, b-(x-a), c)) ):
                ans.append(state)
                return True
        if(b+c<=z):
            if( get_all_states((a, 0, b+c)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((a, b-(z-c), z)) ):
                ans.append(state)
                return True

    if(c>0):
        if(a+c<=x):
            if( get_all_states((a+c, b, 0)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((x, b, c-(x-a))) ):
                ans.append(state)
                return True
        if(b+c<=y):
            if( get_all_states((a, b+c, 0)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((a, y, c-(y-b))) ):
                ans.append(state)
                return True

    return False

initial_state = (12,0,0)
print("Water Jug Problem using Alpha Beta Pruning...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
    print(i)
```

Output

```
PS E:\TY\SEM VI\AI> & C:/Users/varun/AppData/Local/Programs/Python/Python310/python.exe "e:/TY/SEM VI/AI/PRAC_9.py"
Water Jug Problem using Alpha Beta Pruning...

(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
```

Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 07-3-2022	Date of Submission: 07-3-2022

Write a program to solve the Monkey Banana problem by learning concepts.

Suppose the problem is as given below −

- A hungry monkey is in a room, and he is near the door.

- The monkey is on the floor.

- Bananas have been hung from the center of the ceiling of the room.

- There is a block (or chair) present in the room near the window.

- The monkey wants the banana, but cannot reach it.

So if the monkey is clever enough, he can come to the block, drag the block to the center, climb on it, and get the banana. Below are few observations in this case −

- Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both the monkey and the block are on the floor.

- If the block position is not at the center, then monkey can drag it to the center.

- If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.

- When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

Code

```
DO( STATE(MIDDLE, ONBOX, MIDDLE, HASNOT),
   GRAB,
   STATE(MIDDLE, ONBOX, MIDDLE, HAS) ).

DO( STATE(L, ONFLOOR, L, BANANA),
   CLIMB,
   STATE(L, ONBOX, L, BANANA) ).

DO( STATE(L1, ONFLOOR, L1, BANANA),
   PUSH(L1, L2),
   STATE(L2, ONFLOOR, L2, BANANA) ).

DO( STATE(L1, ONFLOOR, BOX, BANANA),
   WALK(L1, L2),
   STATE(L2, ONFLOOR, BOX, BANANA) ).

CANGET(STATE(_, _, _, HAS)).

CANGET(STATE1) :-
```

```prolog
    do(State1, Action, State2),
    canget(State2).

canget(state(_, _, _, has), []).

canget(State1, Plan) :-
    do(State1, Action, State2),
    canget(State2, PartialPlan),
    add(Action, PartialPlan, Plan).

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

Output

Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 28-2-2021	Date of Submission: 28-2-2022

To implement Expert System to diagnose a medical disease.

This medical expert system is used to diagnose the main lung diseases among the patients. The diagnosis is made considering the symptoms that can be seen or felt. This medical expert system helps the doctor or expert in making the appropriate diagnosis of the patient. The lung diseases have many common symptoms and some of them are very much alike. This creates many difficulties for the lungs doctor to reach at a right decision or diagnosis. This expert system can remove these difficulties and it is having knowledge of thirty-two lung diseases. This expert system is implemented in SWI- Prolog.

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) ?").
```

```prolog
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? varun.
varun, do you have a fever (y/n) ?y.
varun, do you have a headache (y/n) ?|: n.
varun, do you have a cough (y/n) ?|: y.
varun, do you have a sneezing (y/n) ?|: y.
varun, do you have a runny_nose (y/n) ?|: y.
varun, probably has measles.
true .
```

Conclusions

This medical expert system is dealing with person's health and an approximate diagnosis of a certain disease is established, this system has a great risk. There may be more lung diseases which are not considered in the system's knowledge base. So, this knowledge base is incomplete, but it can be updated any time with new symptoms and diseases. Symptoms already available in knowledge base are not 100% correct because different doctors have different opinions and there are anomalies in medicines.