Kartik Padave

A022

70362019039

25/01/2022

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:

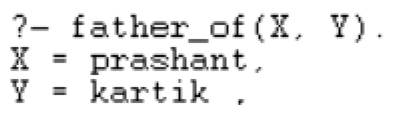
```
parent(prashant, kartik).
parent(priya, kartik).
parent(prashant, chaitanya).
parent(priya, chaitanya).
parent(vinay, varun).
parent(poonam, varun).
parent(vinay, akshata).
parent(poonam, akshata).
parent(nitin, nishita).
parent(sunita, nishita).
parent(nitin, ruchita).
parent(sunita, ruchita).

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).
```

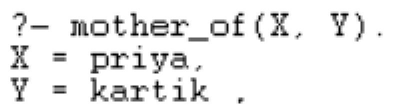
Input:

```
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).
```

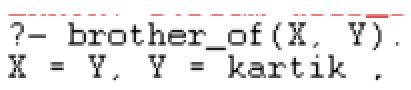
Output:



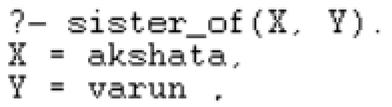
```
?- father_of(X, Y).
X = prashant,
Y = kartik .
```



```
?- 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

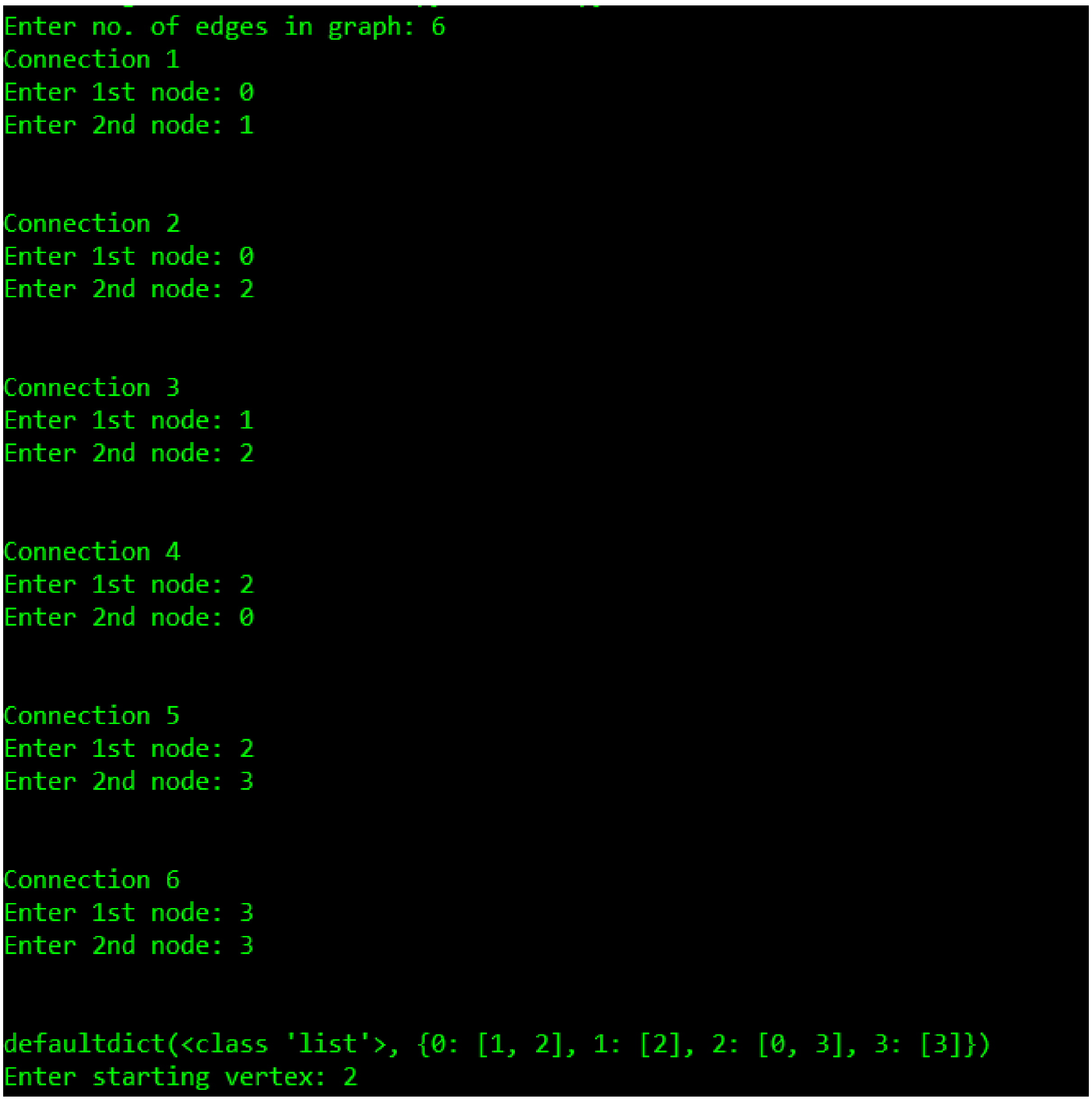
Define:

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:

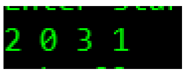
```
import collections

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

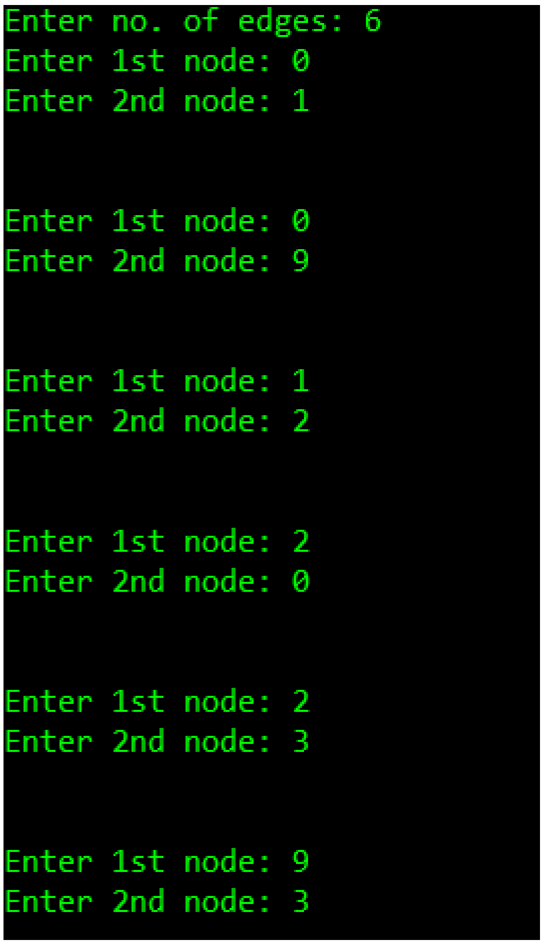
Define:

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:

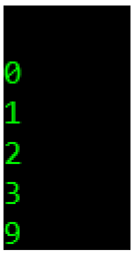
```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    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:

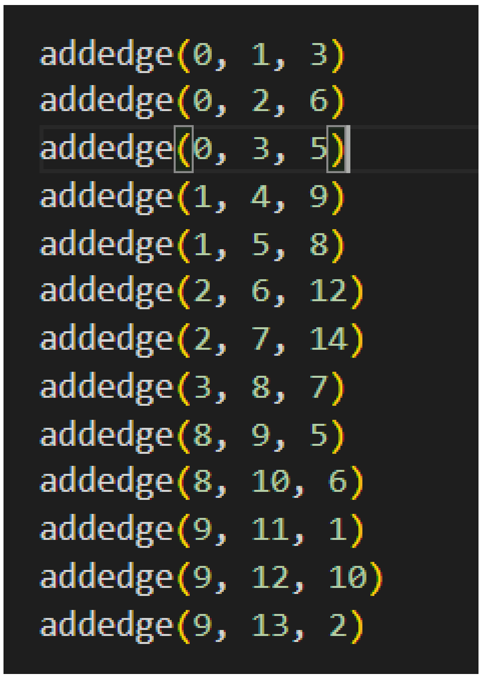
Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it. For this it uses an evaluation function to decide the traversal.

Code:

```python
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)
```

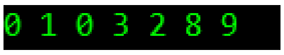
```
source = 0
target = 9
best_first_search(source, target, v)
```

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

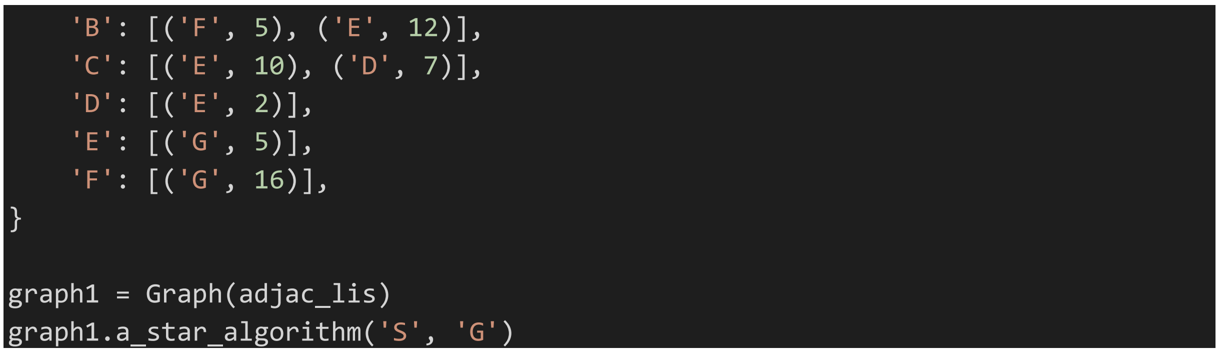
Aim: Write a program to implement A* search algorithm

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.

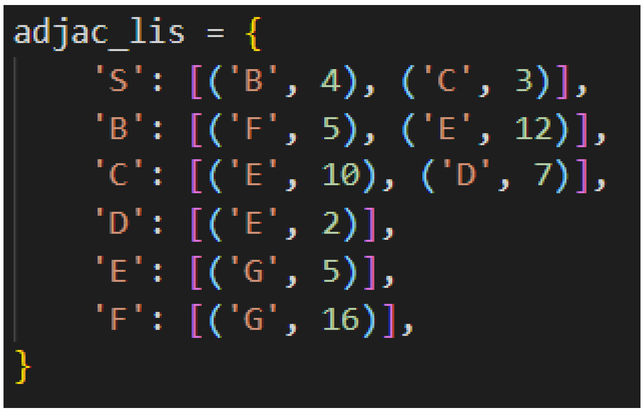
Code:

```python
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)],
```



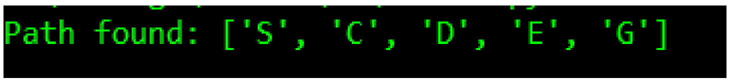
```
    'B': [('F', 5), ('E', 12)],
    'C': [('E', 10), ('D', 7)],
    'D': [('E', 2)],
    'E': [('G', 5)],
    'F': [('G', 16)],
}

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']
```