Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 8-12-2021	Date of Submission: 31-1-2022

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

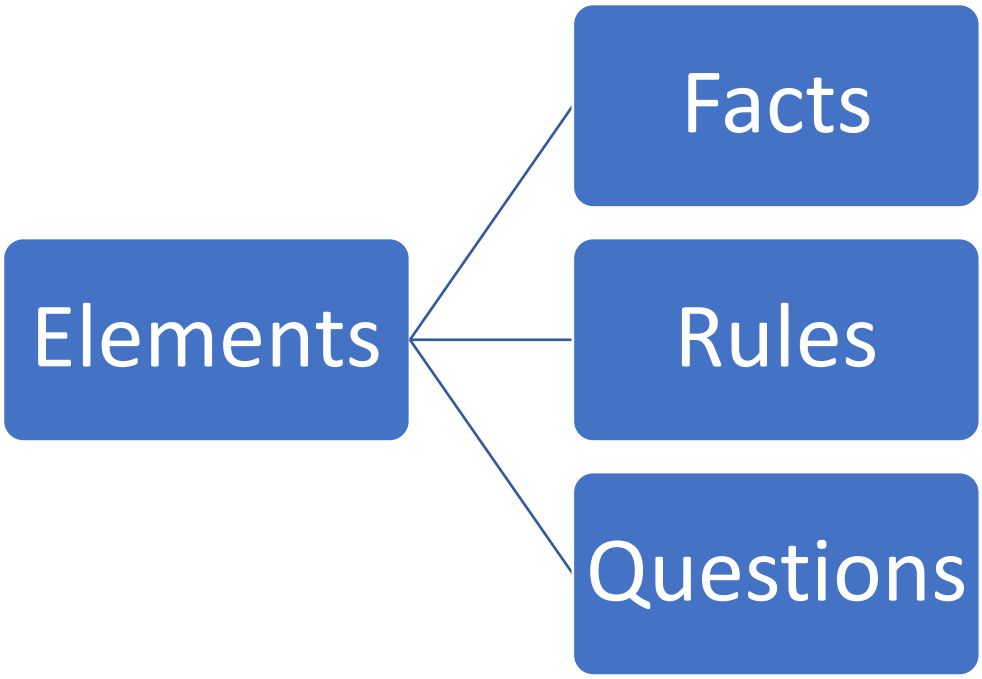
enhanced to include integer value, characters, and operations. The Mechanism of prolog describes the tuples and lists.

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

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:

```
% 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.

?-
```

Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 13-12-2021	Date of Submission: 31-1-2022

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 .
```

Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 20-12-2021	Date of Submission: 31-1-2022

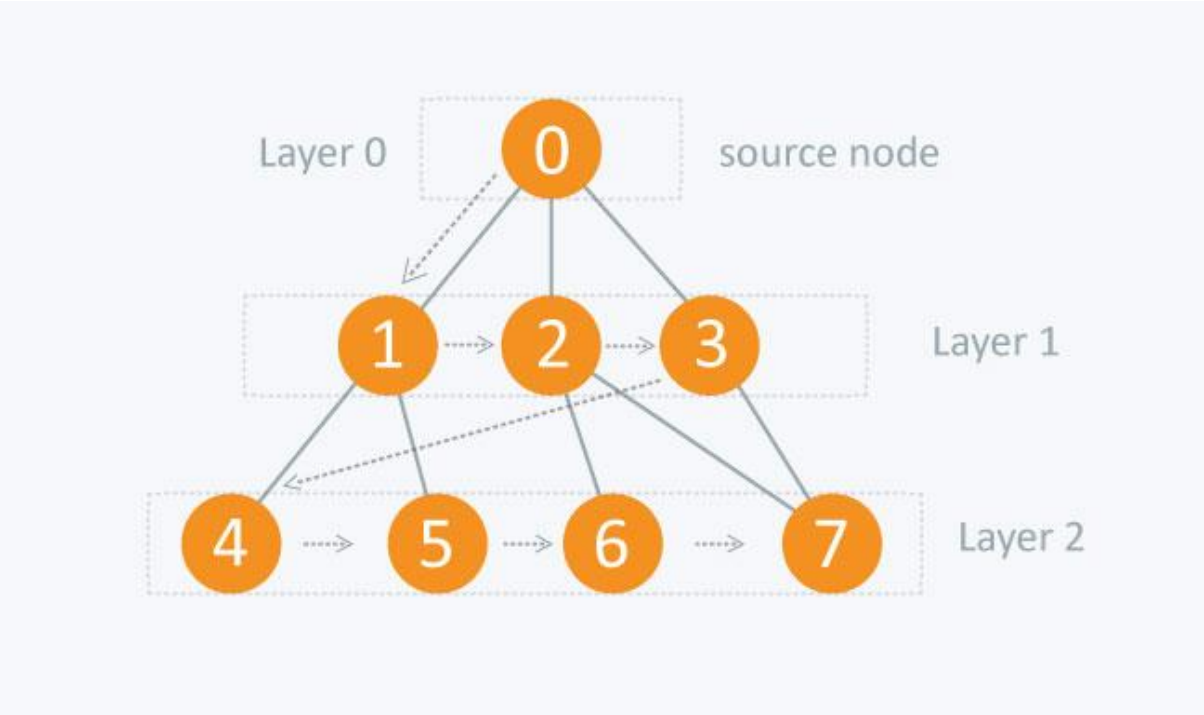
To implement Breadth First Search

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

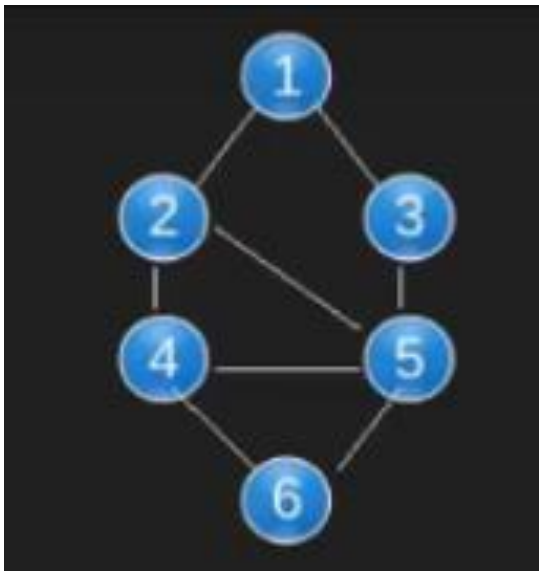
```
#include<stdio.h>
void adj();
void input();
void bfs(int);
int delQue();
void addQue(int);
int v,n,f=0,r=0,visited[10]={0},a[10][10],que[10]={0};
int main()
{
   printf("\tBFS TRAVERSAL\n\n");
        input();
        printf("\nEnter the starting vertex: \n");
        scanf("%d",&v);
        printf("\nBFS 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]);
}
```

```
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.
```

Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 3-1-2021	Date of Submission: 31-1-2022

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.

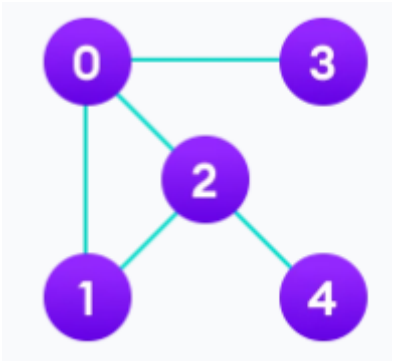
Code

```
#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);
```

```
        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);
        }
}
```

Output

Taking this example




```
"E:\TY\SEM VI\AI\DFS.exe"
Enter the number of nodes: 5
Enter the adjacency matrix of the graph:
0 1 1 1 0
1 0 1 0 0
1 1 0 0 1
1 0 0 0 0
0 0 1 0 0
Enter the source node: 1

The nodes reachable from 1 are:
1 2 3 5 4
Process returned 0 (0x0)   execution time : 75.766 s
Press any key to continue.
```

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

```
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()
```

```
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

```
PS E:\TY\SEM VI\AI> & C:/Users/varun/AppData/Local/Programs/Python/Python310/python.exe "e:/TY/SEM VI/AI/PRAC_3.py"
0 2 3 4
```

Roll. No. A016	Name: Varun Khadayate
Class B.Tech CsBs	Batch: 1
Date of Experiment: 17-1-2021	Date of Submission: 31-1-2022

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

```
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]
```

```
    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
```

```
                        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']
```