

Exp No-1

Date:13/01/2022

Implementation of Toy Problem

Problem Statement:

The N Queen is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other

Algorithm:

- 1) Start in the leftmost column
- 2) If all queens are placed return true
- 3) Try all rows in the current column.
 - 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 the 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.
- 3) If all rows have been tried and nothing worked return false to trigger backtracking.

Code:

```
#include <iostream>
using namespace std;
const int n = 8;
bool isSafe(int array[][n],int x,int y,int n)
{
    for(int row=0;row<x;row++)
    {
        if(array[row][y]==1)
```

```

        return false;
    }
    int row=x;
    int col=y;
    while(row>=0&&col>=0)
    {
        if(array[row][col]==1)
            return false;

        row--;
        col--;
    }
    row=x;
    col=y;
    while(row>=0&&col<n)
    {
        if(array[row][col]==1)
            return false;

        row--;
        col++;
    }
    return true;
}

```

```

bool nqueen(int array[][n],int x,int n)
{
    if(x>=n)
        return true;
    for(int col=0;col<n;col++)
    {
        if(isSafe(array,x,col,n)){
            array[x][col]=1;

```

```

        if(nqueen(array,x+1,n))
            return true;
    }
    array[x][col]=0;
}
return false;
}

```

```

int main() {

    int arr[n][n];
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
            arr[i][j]=0;
    }
    if(nqueen(arr,0,n)){
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
                cout<<arr[i][j]<<" ";
            cout<<endl;
        }
    }
    return 0;
}

```

INPUT /OUTPUT

- 1) For 4 queens

Output:

```
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```

2) For 8 queen

Output:

```
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
```

Here 1 represents queen is placed.

Time Complexity : $O(n^2)$

Result:

N-queen problem is implemented.

EXP NO : 2

Date:24/01/2022

Developing agent programs for real world problems

Problem Statement:

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

Algorithm:

- 1) Consider city 1 as the starting and ending point.
- 2) Generate all $(n-1)!$ Permutations of cities.
- 3) Calculate cost of every permutation and keep track of minimum cost permutation.
- 4) Return the permutation with minimum cost.

Code:

```
#include<iostream>
```

```
using namespace std;
```

```
int ary[10][10],completed[10],n,cost=0;
```

```
void takeInput()
```

```
{
```

```
int i,j;
```

```
cin>>n;
```

```
for(i=0;i < n;i++)
```

```
{
```

```
for( j=0;j < n;j++)
```

```
cin>>ary[i][j];
```

```
completed[i]=0;
```

```
}
```

```
cout<<"\n\nThe cost list is:";
```

```
for( i=0;i < n;i++)
```

```
{
```

```
cout<<"\n";
```

```
for(j=0;j < n;j++)
```

```
cout<<"\t"<<ary[i][j];
```

```
}
```

```
}
```

```
int least(int c)
```

```
{
```

```
int i,nc=999;
```

```
int min=999,kmin;
```

```
for(i=0;i < n;i++)
```

```
{
```

```
if((ary[c][i]!=0)&&(completed[i]==0))
```

```
if(ary[c][i]+ary[i][c] < min)
```

```
{
```

```
min=ary[i][0]+ary[c][i];
```

```
kmin=ary[c][i];
```

```
nc=i;
```

```
}
```

```
}
```

```
if(min!=999)
```

```
cost+=kmin;
```

```
return nc;
```

```
}
```

```
void mincost(int city)
```

```
{
```

```
int i,ncity;
```

```
completed[city]=1;
```

```
cout<<city+1<<"--->";
```

```
ncity=least(city);
```

```
if(ncity==999)
```

```
{
```

```
ncity=0;
```

```
cout<<ncity+1;
```

```
cost+=ary[city][ncity];
```



```
return;
```

```
}
```

```
mincost(ncity);
```

```
}
```

```
int main()
```

```
{
```

```
takeInput();
```

```
cout<<"\n\nThe Path is:\n";
```

```
mincost(0); //passing 0 because starting vertex
```

```
cout<<"\n\nMinimum cost is "<<cost;
```

```
return 0;
```

```
}
```

Input

4

0 10 15 20

10 0 35 25

15 35 0 30

20 25 30 0

Output

The cost list is:

0	10	15	20
10	0	35	25
15	35	0	30
20	25	30	0

The Path is:

1--2---3---4---1

Minimum cost is 95

Time Complexity: $O(n^4)$

Result : Travel-Salesman Problem is solved.

EXP-3

DATE:08/02/2022

CSP - Graph Coloring Problem

Problem Statement

The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color.

Algorithm:

1. Graph / map coloring problems are those where the nodes are assigned colors such that the adjacent connected nodes / regions don't have the same color assigned.
2. At the same time, it is required to use the minimum number of colors possible – called the chromatic number.
3. Start by coloring the first node with a color.
4. Color the subsequent connected nodes with a different color.
5. Check at every step that it satisfies the condition.

Program:

```
class Graph:
    def __init__(self, edges, N):
        self.adj = [[] for _ in range(N)]
        for (src, dest) in edges:
            self.adj[src].append(dest)
            self.adj[dest].append(src)

def colorGraph(graph):
    result = {}
    for u in range(N):
        assigned = set([result.get(i) for i in graph.adj[u] if i in result])
        color = 1
        for c in assigned:
            if color != c:
                break
        color = color + 1
        result[u] = color
    for v in range(N):
        print("Color assigned to vertex", v, "is", colors[result[v]])

if __name__ == '__main__':
    colors = ["", "BLUE", "GREEN", "RED", "YELLOW", "ORANGE"]
```

```
edges = [(0,1), (0,3), (0,4), (1,0), (1,4), (1,2), (2,1), (2,3), (3,0), (3,4), (3,5), (4,0), (4,1), (4,2), (4,3), (4,5), (4,6), (5,3), (5,6), (5,4), (6,4), (6,5)]
N = 7
graph = Graph(edges, N)
colorGraph(graph)
```

Output:

Output:

```
➞ Color assigned to vertex 0 is BLUE
Color assigned to vertex 1 is GREEN
Color assigned to vertex 2 is BLUE
Color assigned to vertex 3 is GREEN
Color assigned to vertex 4 is RED
Color assigned to vertex 5 is BLUE
Color assigned to vertex 6 is GREEN
```

Result: Thus, Graph coloring has been successfully implemented.

EXP-4

Date: 22/02/2022

Implementation and Analysis of DFS and BFS

Problem Statement: Given a grid where each cell can take '#' or '.' Goal is to find the longest path that one can go through in this grid using DFS, given a binary tree find the depth of binary tree using BFS.

Algorithm (DFS):

1. Create a recursive function that takes the index of node and a visited array.
2. Mark the current node as visited and print the node.
3. Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.

Algorithm (BFS):

1. Start with node and push that node into the queue.
2. Pop n nodes from the queue and process it and push all its child nodes into the queue
3. Process all the nodes present in the queue in similar manner until queue is empty

Code(Bfs)

```
#include<iostream>

#include<bits/stdc++.h>

using namespace std;

int main(){

    int V,e;

    cout<<"Enter no of vertices: "<<endl;

    cin>>V;

    cout<<"Enter no of edges: "<<endl;

    cin>>e;

    vector<int>adj[V];

    cout<<"Create graph: "<<endl;

    for(int i=0;i<e;i++){
```

```

    int start,end;

    cin>>start>>end;

    adj[start].push_back(end);
}

vector<int>ans;

bool visited[V]={false};

queue<int>q1;

q1.push(0);

while(!q1.empty()){

    int node=q1.front();

    q1.pop();

    visited[node]=true;

    ans.push_back(node);

    for(auto it:adj[node]){

        if(visited[it]==false){

            q1.push(it);

            visited[it]=true;

        }

    }

}

cout<<"Bfs: "<<endl;

for(auto it:ans)

    cout<<it<<" ";

}

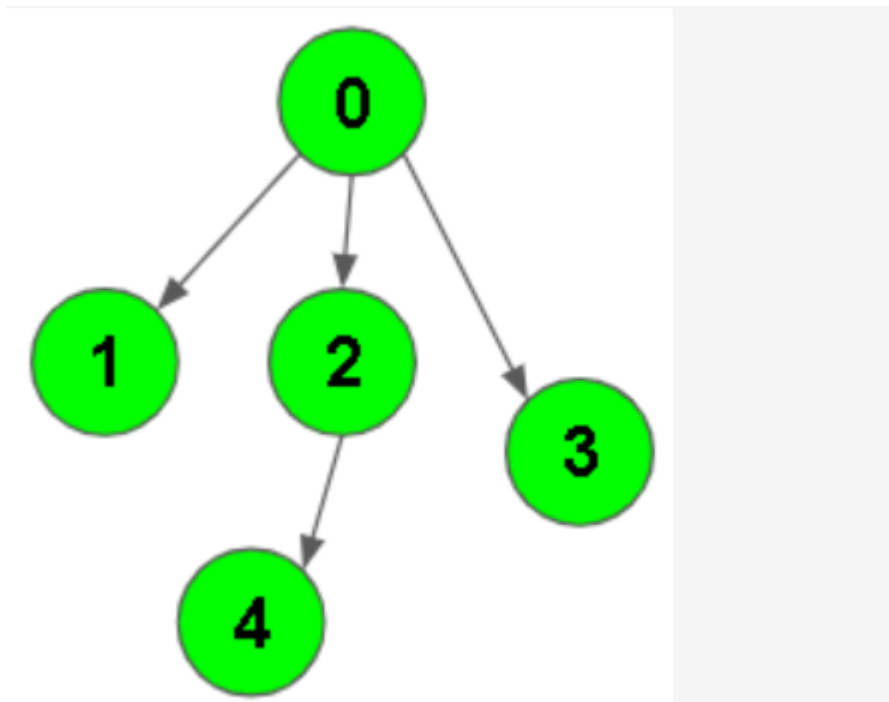
```

Input:

```

Running /home/ubuntu/environment/RA1911003010305/bfs.cpp
Enter no of vertices:
5
Enter no of edges:
4
Create graph:
0 1
0 2
0 3
2 4

```



Output

```
Running /home/ubuntu/environment/RA1911003010305/bfs.cpp
Enter no of vertices:
5
Enter no of edges:
4
Create graph:
0 1
0 2
0 3
2 4
Bfs:
0 1 2 3 4
Process exited with code: 0
```

For DFS

Code:

```
#include<iostream>
```

```
#include<bits/stdc++.h>
```

```
using namespace std;
```

```
void dfs(int node,vector<bool>&visited,vector<int>adj[],vector<int>&ans){
    if(visited[node]==true)
        return;
```

```

        ans.push_back(node);
        visited[node]=true;
        for(int i=0;i<adj[node].size();i++){
            if(visited[adj[node][i]]==false)
                dfs(adj[node][i],visited,adj,ans);
        }

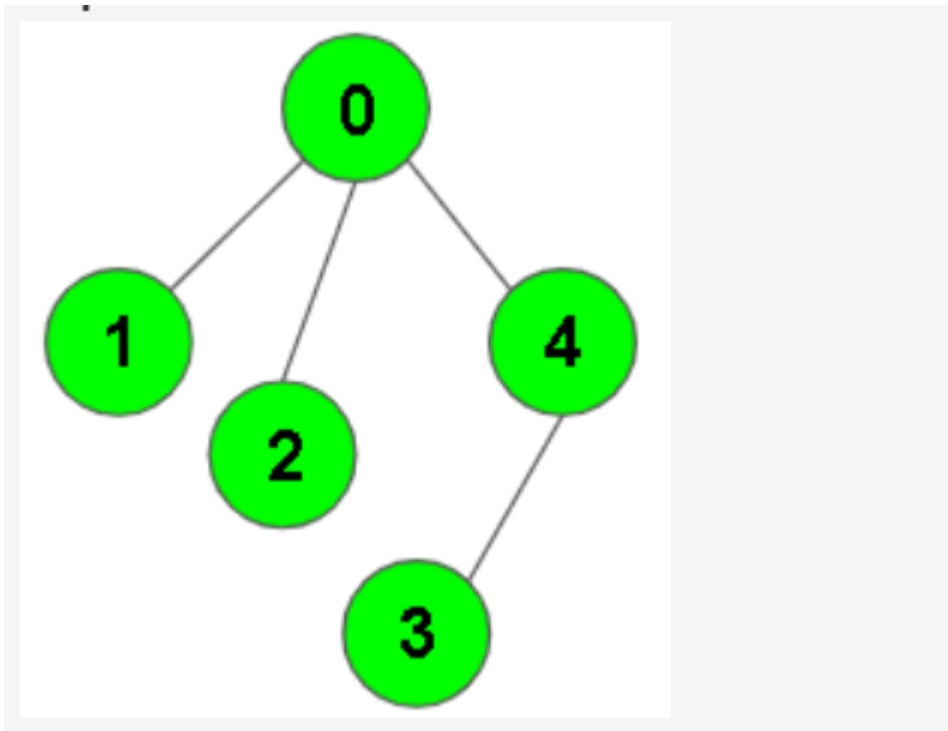
    }

int main(){
    int V,e;
    cout<<"Enter no of vertices: "<<endl;
    cin>>V;
    cout<<"Enter no of edges: "<<endl;
    cin>>e;
    vector<int>adj[V];
    cout<<"Create graph: "<<endl;
    for(int i=0;i<e;i++){
        int start,end;
        cin>>start>>end;
        adj[start].push_back(end);
    }
    vector<bool>visited(V,false);
    vector<int>ans;
    dfs(0,visited,adj,ans);
    cout<<"dfs: "<<endl;
    for(auto it:ans)
        cout<<it<<" ";
}

```

Input:


```
Running /home/ubuntu/environment/RA1911003010305/dfs.cpp
Enter no of vertices:
5
Enter no of edges:
4
Create graph:
0 1
0 2
0 4
4 3
```



Output:

```
Running /home/ubuntu/environment/RA1911003010305/dfs.cpp
Enter no of vertices:
5
Enter no of edges:
4
Create graph:
0 1
0 2
0 4
4 3
dfs:
0 1 2 4 3
```

R
came c

Time complexity: $O(V+E)$

Result:

BFS and DFS Algorithms executed.

Developing Best first search and A star algorithm for real world

Problem Statement: Given a 2D grid having several obstacles, start from source cell and find a path to goal cell using A star algorithm.

Algorithm (A*):

1. Initialize the open list
2. Initialize the closed list, put the starting node on the open list (you can leave its f at zero)
3. while the open list is not empty
 - a. Find the node with the least f on the open list, call it 'q'
 - b. Pop q off the open list
 - c. generate q's 8 successors and set their parents to q
 - d. for each successor
 - i. if successor is the goal, stop search $\text{successor.g} = \text{q.g} + \text{distance between successor and q}$ $\text{successor.h} = \text{distance from goal to successor}$ $\text{successor.f} = \text{successor.g} + \text{successor.h}$
 - ii. if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
 - iii. if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list
 - b. Push q on the closed list

Algorithm (Best first search):

1. Take an input of the maze in binary format.

2. Taking the starting point, find all adjacent paths that can be taken.
3. Keep traversing through the array while taking the adjacent cells closest to the destination while avoiding cells with value 0.
4. If the final point is reached, save the length of the path.
5. Compare lengths of all paths that reach the destination and print the length of the shortest path.

Code:

Best Search

```
#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> pi;

vector<vector<pi> > graph;

// Function for adding edges to graph
void addedge(int x, int y, int cost)
{
    graph[x].push_back(make_pair(cost, y));
    graph[y].push_back(make_pair(cost, x));
}

void best_first_search(int source, int target, int n)
{
    vector<bool> visited(n, false);
```

```

// MIN HEAP priority queue
priority_queue<pi, vector<pi>, greater<pi> > pq;

// sorting in pq gets done by first value of pair
pq.push(make_pair(0, source));

int s = source;
visited[s] = true;
while (!pq.empty()) {
    int x = pq.top().second;
    // Displaying the path having lowest cost
    cout << x << " ";

    pq.pop();
    if (x == target)
        break;

    for (int i = 0; i < graph[x].size(); i++) {
        if (!visited[graph[x][i].second]) {
            visited[graph[x][i].second] = true;
            pq.push(make_pair(graph[x][i].first, graph[x][i].second));
        }
    }
}
}

```

```

// Driver code to test above methods
int main()
{
    // No. of Nodes
    int v;

    cout<<"Enter no of nodes :"<<endl;
    cin>>v;

    graph.resize(v);
}

```

```

    int edge;

    cout<<"Enter no of edge: "<<endl;
    cin>>edge;

    // The nodes shown in above example(by alphabets) are
    // implemented using integers addedge(x,y,cost);
    for(int i=0;i<edge;i++){

        int u,v,cost;

        cin>>u>>v>>cost;

        addedge(u, v, cost);

    }

    int source,target;

    cout<<"Enter source and target : "<<endl;
    cin>>source>>target;

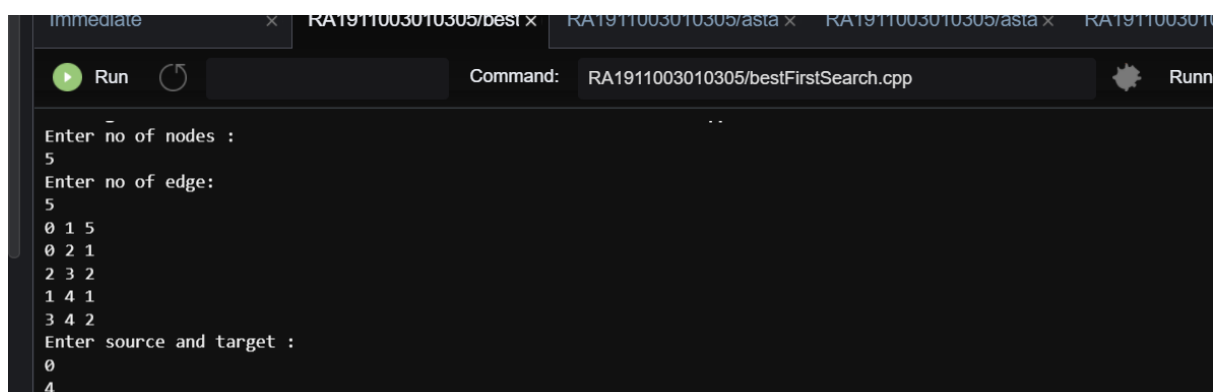

    // Function call
    best_first_search(source, target, v);


    return 0;

}

```

Input:



```

Enter no of nodes :
5
Enter no of edge:
5
0 1 5
0 2 1
2 3 2
1 4 1
3 4 2
Enter source and target :
0
4

```

Output



```

0 2 3 4
Process exited with code: 0

```

A* Algorithm Code

```
def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)
    closed_set = set()

    g = {} #store distance from starting node
    parents = {} # parents contains an adjacency map of all nodes

    #distance of starting node from itself is zero
    g[start_node] = 0

    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None

        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v

        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
```

```
open_set.add(m)
parents[m] = n
g[m] = g[n] + weight
```

```
#for each node m,compare its distance from start i.e g(m) to the
#from start through n node
```

```
else:
```

```
    if g[m] > g[n] + weight:
        #update g(m)
        g[m] = g[n] + weight
        #change parent of m to n
        parents[m] = n
```

```
#if m in closed set,remove and add to open
```

```
if m in closed_set:
    closed_set.remove(m)
    open_set.add(m)
```

```
if n == None:
```

```
    print('Path does not exist!')
```

```
    return None
```

```
# if the current node is the stop_node
```

```
# then we begin reconstructin the path from it to the start_node
```

```
if n == stop_node:
```

```
    path = []
```

```
while parents[n] != n:
```

```
    path.append(n)
```

```
    n = parents[n]
```

```
path.append(start_node)
```

```
path.reverse()
```

```
print('Path found: {}'.format(path))
```

```
return path
```

```
# remove n from the open_list, and add it to closed_list
```

```
# because all of his neighbors were inspected
```

```
open_set.remove(n)
```

```
closed_set.add(n)
```

```
print('Path does not exist!')
```

```
return None
```

```
#define fuction to return neighbor and its distance
```

```
#from the passed node
```

```
def get_neighbors(v):
```

```
    if v in Graph_nodes:
```

```
        return Graph_nodes[v]
```

```
    else:
```

```
        return None
```

```
#for simplicity we ll consider heuristic distances given
```

```
#and this function returns heuristic distance for all nodes
```

```
def heuristic(n):
```

```
    H_dist = {
```

```
        'A': 11,
```

```
        'B': 6,
```

```
        'C': 99,
```



```

        'D': 1,

        'E': 7,

        'G': 0,

    }

    return H_dist[n]

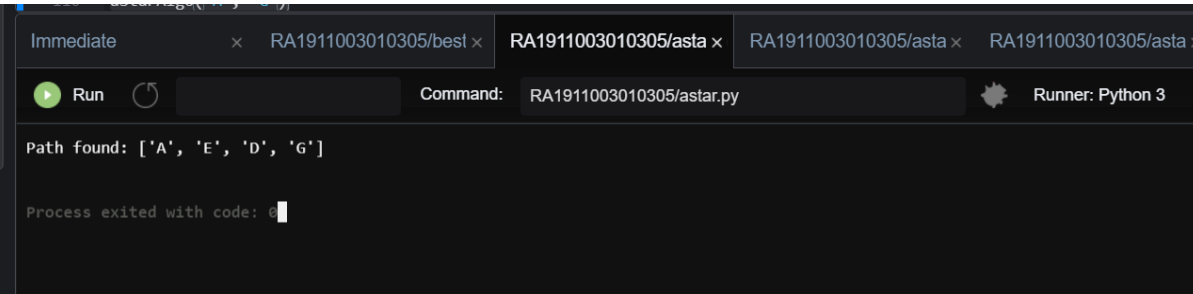
#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}

aStarAlgo('A', 'G')

```

Input/Output



```

RA1911003010305/asta x RA1911003010305/asta x RA1911003010305/asta x RA1911003010305/asta x
Run Command: RA1911003010305/astar.py Runner: Python 3
Path found: ['A', 'E', 'D', 'G']
Process exited with code: 0

```

Result : The path in grid with obstacles was found using A star algorithm and BFS and was displayed as result in terminal.

Exp-6

DATE:15/03/2022

Implementation of uncertain methods for an application

Problem Statement: To implement uncertain methods for an application (to calculate membership of certain values based on user's input and display the same) using Fuzzy logic

Algorithm:

1. Define Non Fuzzy Inputs with Fuzzy Sets. The non-fuzzy inputs are numbers from a certain range, and find how to represent those non-fuzzy values with fuzzy sets.
2. Locate the input, output, and state variables of the plane under consideration.
3. Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.
4. Obtain the membership function for each fuzzy subset.
5. Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and output of fuzzy subsets on the other side, thereby forming the rule base.
6. Choose appropriate scaling factors for the input and output variables for normalizing the variables between [0, 1] and [-1, 1] interval.
7. Carry out the fuzzification process.
8. Identify the output contributed from each rule using fuzzy approximate reasoning.
9. Combine the fuzzy outputs obtained from each rule.

Finally, apply defuzzification to form a crisp output

CODE:

```
#include <iostream>
```

```
#include <cmath>
```

```
#include <cstring>
```

```
const double cdMinimumPrice =0;
```

```
const double cdMaximumPrice =70;
```

```

using namespace std;

class CFuzzyFunction
{
protected :

    double dLeft, dRight;

    char  cType;

    char* sName;

public:

    CFuzzyFunction(){};

    virtual ~CFuzzyFunction(){ delete [] sName; sName=NULL;}

    virtual void

    setInterval(double l,

                double r)

    {dLeft=l; dRight=r;}

    virtual void

    setMiddle( double dL=0,

              double dR=0)=0;

    virtual void

    setType(char c)

```

```
{ cType=c;}
```

```
virtual void
```

```
setName(const char* s)
```

```
{
```

```
    sName = new char[strlen(s)+1];
```

```
    strcpy(sName,s);
```

```
}
```

```
bool
```

```
isDotInInterval(double t)
```

```
{
```

```
    if((t>=dLeft)&&(t<=dRight)) return true; else return false;
```

```
}
```

```
char getType(void)const{ return cType;}
```

```
void
```

```
getName() const
```

```
{
```

```
    cout<<sName<<endl;
```

```
}
```

```
virtual double getValue(double t)=0;
```

```
};
```

```
class CTriangle : public CFuzzyFunction
```

```
{
```

```
private:
```

```
    double dMiddle;
```

```
public:
```

```
    void
```

```
    setMiddle(double dL, double dR)
```

```
{
```

```
    dMiddle=dL;
```

```
}
```

```
    double
```

```
    getValue(double t)
```

```
{
```

```
    if(t<=dLeft)
```

```
        return 0;
```

```
    else if(t<dMiddle)
```

```
        return (t-dLeft)/(dMiddle-dLeft);
```

```
    else if(t==dMiddle)
```

```
        return 1.0;
```

```
        else if(t<dRight)
            return (dRight-t)/(dRight-dMiddle);
        else
            return 0;
    }
};
```

```
class CTrapezoid : public CFuzzyFunction
{
private:
    double dLeftMiddle, dRightMiddle;

public:
    void
    setMiddle(double dL, double dR)
    {
        dLeftMiddle=dL; dRightMiddle=dR;
    }

    double
    getValue(double t)
    {
        if(t<=dLeft)
```

```

        return 0;

        else if(t<dLeftMiddle)

            return (t-dLeft)/(dLeftMiddle-dLeft);

        else if(t<=dRightMiddle)

            return 1.0;

        else if(t<dRight)

            return (dRight-t)/(dRight-dRightMiddle);

        else

            return 0;

    }

};

int
main(void)
{
    CFuzzyFunction *FuzzySet[3];

    FuzzySet[0] = new CTrapezoid;
    FuzzySet[1] = new CTriangle;
    FuzzySet[2] = new CTrapezoid;

    FuzzySet[0]->setInterval(-5,30);
    FuzzySet[0]->setMiddle(0,20);

```

```

FuzzySet[0]->setType('r');
FuzzySet[0]->setName("low_price");

FuzzySet[1]->setInterval(25,45);
FuzzySet[1]->setMiddle(35,35);
FuzzySet[1]->setType('t');
FuzzySet[1]->setName("good_price");

FuzzySet[2]->setInterval(40,75);
FuzzySet[2]->setMiddle(50,70);
FuzzySet[2]->setType('r');
FuzzySet[2]->setName("to_expensive");

double dValue;

    do
{
    cout<<"\nInput the value->"; cin>>dValue;

    if(dValue<cdMinimumPrice) continue;
    if(dValue>cdMaximumPrice) continue;

    for(int i=0; i<3; i++)
    {
        cout<<"\nThe dot="<<dValue<<endl;

```



```

        if(FuzzySet[i]->isDotInInterval(dValue))
            cout<<"In the interval";
        else
            cout<<"Not in the interval";

        cout<<endl;

        cout<<"The name of function is"<<endl;

        FuzzySet[i]->getName();

        cout<<"and the membership is=";

        cout<<FuzzySet[i]->getValue(dValue);

    }

}

while(true);

return EXIT_SUCCESS;

}

```

INPUT

The screenshot shows a C++ IDE with a terminal window. The terminal displays the command prompt 'Input the value->9'. The IDE interface includes a file explorer on the left, a code editor in the center, and a terminal at the bottom. The terminal output shows the program running successfully.

OUTPUT

```
Stop [refresh] Command: RA1911003010305/fuzzyLogic.cpp Runner: C++ CWD En

Running /home/ubuntu/environment/RA1911003010305/fuzzyLogic.cpp

Input the value->9

The dot=9
In the interval
The name of function is
low_price
and the membership is=1
The dot=9
Not in the interval
The name of function is
good_price
and the membership is=0
The dot=9
Not in the interval
The name of function is
to_expensive
and the membership is=0
```

Result: Implementation of uncertain methods for an application (to calculate membership of certain values) is successfully implemented.

EXP-7

Date:15/03/2022

Implementation of unification and resolution for real world problems.

Problem Statement: Develop a program to unify expressions and direct the output of resolution to output.txt after taking input from input.txt file in same directory .

Algorithm (unification):

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

- a) If Ψ_1 or Ψ_2 are identical, then return NIL.
- b) Else if Ψ_1 is a variable,
 - a. then if Ψ_1 occurs in Ψ_2 , then return FAILURE
 - b. Else return $\{ (\Psi_2 / \Psi_1) \}$.
- c) Else if Ψ_2 is a variable,
 - a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
 - b. Else return $\{ (\Psi_1 / \Psi_2) \}$.
- d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For $i=1$ to the number of elements in Ψ_1 .

- a) Call Unify function with the i th element of Ψ_1 and i th element of Ψ_2 , and put the result into S.
- b) If $S = \text{failure}$ then returns Failure
- c) If $S \neq \text{NIL}$ then do,
 - a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S,
SUBST). Step.6: Return SUBST.

Algorithm (resolution):

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

Unification:

Code:

```
def get_index_comma(string):  
    """  
    Return index of commas in string  
    """  
  
    index_list = list()  
    # Count open parentheses  
    par_count = 0  
  
    for i in range(len(string)):  
        if string[i] == ',' and par_count == 0:  
            index_list.append(i)  
        elif string[i] == '(':  
            par_count += 1  
        elif string[i] == ')':  
            par_count -= 1
```

```
return index_list
```

```
def is_variable(expr):
```

```
    """
```

```
    Check if expression is variable
```

```
    """
```

```
    for i in expr:
```

```
        if i == '(':
```

```
            return False
```

```
    return True
```

```
def process_expression(expr):
```

```
    """
```

```
    input: - expression:
```

```
           'Q(a, g(x, b), f(y))'
```

```
    return: - predicate symbol:
```

```
            Q
```

```
            - list of arguments
```

```
            ['a', 'g(x, b)', 'f(y)']
```

```
    """
```

```
    # Remove space in expression
```

```
    expr = expr.replace(' ', '')
```

```
    # Find the first index == '('
```

```
    index = None
```

```
    for i in range(len(expr)):
```

```
        if expr[i] == '(':
```

```
index = i
```

```
break
```

```
# Return predicate symbol and remove predicate symbol in expression
```

```
predicate_symbol = expr[:index]
```

```
expr = expr.replace(predicate_symbol, "")
```

```
# Remove '(' in the first index and ')' in the last index
```

```
expr = expr[1:len(expr) - 1]
```

```
# List of arguments
```

```
arg_list = list()
```

```
# Split string with commas, return list of arguments
```

```
indices = get_index_comma(expr)
```

```
if len(indices) == 0:
```

```
    arg_list.append(expr)
```

```
else:
```

```
    arg_list.append(expr[:indices[0]])
```

```
    for i, j in zip(indices, indices[1:]):
```

```
        arg_list.append(expr[i + 1:j])
```

```
    arg_list.append(expr[indices[len(indices) - 1] + 1:])
```

```
return predicate_symbol, arg_list
```

```
def get_arg_list(expr):
```

```
    """
```

```
    input: expression:
```

```
        'Q(a, g(x, b), f(y))'
```

```
    return: full list of arguments:
```

```
['a', 'x', 'b', 'y']
```

```
"""
```

```
_, arg_list = process_expression(expr)
```

```
flag = True
```

```
while flag:
```

```
    flag = False
```

```
    for i in arg_list:
```

```
        if not is_variable(i):
```

```
            flag = True
```

```
            _, tmp = process_expression(i)
```

```
            for j in tmp:
```

```
                if j not in arg_list:
```

```
                    arg_list.append(j)
```

```
            arg_list.remove(i)
```

```
    return arg_list
```

```
def check_occurs(var, expr):
```

```
    """
```

```
    Check if var occurs in expr
```

```
    """
```

```
    arg_list = get_arg_list(expr)
```

```
    if var in arg_list:
```

```
        return True
```

```
    return False
```

```

def unify(expr1, expr2):
    """
    Unification Algorithm

    Step 1: If  $\Psi_1$  or  $\Psi_2$  is a variable or constant, then:
        a, If  $\Psi_1$  or  $\Psi_2$  are identical, then return NULL.
        b, Else if  $\Psi_1$  is a variable:
            - then if  $\Psi_1$  occurs in  $\Psi_2$ , then return False
            - Else return ( $\Psi_2 / \Psi_1$ )
        c, Else if  $\Psi_2$  is a variable:
            - then if  $\Psi_2$  occurs in  $\Psi_1$ , then return False
            - Else return ( $\Psi_1 / \Psi_2$ )
        d, Else return False

    Step 2: If the initial Predicate symbol in  $\Psi_1$  and  $\Psi_2$  are not same, then return False.

    Step 3: IF  $\Psi_1$  and  $\Psi_2$  have a different number of arguments, then return False.

    Step 4: Create Substitution list.

    Step 5: For i=1 to the number of elements in  $\Psi_1$ .
        a, Call Unify function with the ith element of  $\Psi_1$  and ith element of  $\Psi_2$ , and put the result
        into S.
        b, If S = False then returns False
        c, If S  $\neq$  Null then append to Substitution list

    Step 6: Return Substitution list.
    """

    # Step 1:
    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False

    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):

```



```

        return False
    else:
        tmp = str(expr2) + '/' + str(expr1)
        return tmp
elif not is_variable(expr1) and is_variable(expr2):
    if check_occurs(expr2, expr1):
        return False
    else:
        tmp = str(expr1) + '/' + str(expr2)
        return tmp
else:
    predicate_symbol_1, arg_list_1 = process_expression(expr1)
    predicate_symbol_2, arg_list_2 = process_expression(expr2)

    # Step 2
    if predicate_symbol_1 != predicate_symbol_2:
        return False

    # Step 3
    elif len(arg_list_1) != len(arg_list_2):
        return False
    else:
        # Step 4: Create substitution list
        sub_list = list()

        # Step 5:
        for i in range(len(arg_list_1)):
            tmp = unify(arg_list_1[i], arg_list_2[i])

            if not tmp:
                return False
            elif tmp == 'Null':

```

```

        pass
    else:
        if type(tmp) == list:
            for j in tmp:
                sub_list.append(j)
        else:
            sub_list.append(tmp)

# Step 6
return sub_list

if __name__ == '__main__':
    # Data 1
    f1 = 'p(b(A), X, f(g(Z)))'
    f2 = 'p(Z, f(Y), f(Y))'

    # Data 2
    # f1 = 'Q(a, g(x, a), f(y))'
    # f2 = 'Q(a, g(f(b), a), x)'

    # Data 3
    # f1 = 'Q(a, g(x, a, d), f(y))'
    # f2 = 'Q(a, g(f(b), a), x)'

    result = unify(f1, f2)
    if not result:
        print('Unification failed!')
    else:
        print('Unification successfully!')
        print(result)

```

Input/Output



```
23 Check if expression is variable
24 """
25
26 for i in expr:
```

9:18 Python

Immediate x RA191100301030 x RA191100301030 x RA191100301030 x RA191100301030 x RA191100301030 x RA191100301030 x RA191100301030 x

Run Command: RA1911003010305/unification.py Runner: Python 3 CWD

Unification successfully!
['b(A)/Z', 'f(Y)/X', 'g(Z)/Y']

Process exited with code: 0

Resolution:

Code

import copy

import time

class Parameter:

variable_count = 1

def __init__(self, name=None):

if name:

self.type = "Constant"

self.name = name

else:

self.type = "Variable"

self.name = "v" + str(Parameter.variable_count)

Parameter.variable_count += 1

def isConstant(self):

return self.type == "Constant"

def unify(self, type_, name):

self.type = type_

```
self.name = name
```

```
def __eq__(self, other):  
    return self.name == other.name
```

```
def __str__(self):  
    return self.name
```

```
class Predicate:
```

```
def __init__(self, name, params):  
    self.name = name  
    self.params = params
```

```
def __eq__(self, other):  
    return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))
```

```
def __str__(self):  
    return self.name + "(" + ",".join(str(x) for x in self.params) + ")"
```

```
def getNegatedPredicate(self):  
    return Predicate(negatePredicate(self.name), self.params)
```

```
class Sentence:
```

```
    sentence_count = 0
```

```
def __init__(self, string):  
    self.sentence_index = Sentence.sentence_count  
    Sentence.sentence_count += 1  
    self.predicates = []  
    self.variable_map = {}
```

```
local = {}
```

```
for predicate in string.split(" | "):
```

```
    name = predicate[:predicate.find("(")]
```

```
    params = []
```

```
    for param in predicate[predicate.find("(") + 1: predicate.find(")"]].split(", "):
```

```
        if param[0].islower():
```

```
            if param not in local: # Variable
```

```
                local[param] = Parameter()
```

```
                self.variable_map[local[param].name] = local[param]
```

```
                new_param = local[param]
```

```
            else:
```

```
                new_param = Parameter(param)
```

```
                self.variable_map[param] = new_param
```

```
    params.append(new_param)
```

```
self.predicates.append(Predicate(name, params))
```

```
def getPredicates(self):
```

```
    return [predicate.name for predicate in self.predicates]
```

```
def findPredicates(self, name):
```

```
    return [predicate for predicate in self.predicates if predicate.name == name]
```

```
def removePredicate(self, predicate):
```

```
    self.predicates.remove(predicate)
```

```
    for key, val in self.variable_map.items():
```

```
        if not val:
```

```
            self.variable_map.pop(key)
```

```
def containsVariable(self):
```

```
    return any(not param.isConstant() for param in self.variable_map.values())
```

```
def __eq__(self, other):
```

```
    if len(self.predicates) == 1 and self.predicates[0] == other:
```

```
        return True
```

```
    return False
```

```
def __str__(self):
```

```
    return "".join([str(predicate) for predicate in self.predicates])
```

```
class KB:
```

```
    def __init__(self, inputSentences):
```

```
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
```

```
        self.sentences = []
```

```
        self.sentence_map = {}
```

```
    def prepareKB(self):
```

```
        self.convertSentencesToCNF()
```

```
        for sentence_string in self.inputSentences:
```

```
            sentence = Sentence(sentence_string)
```

```
            for predicate in sentence.getPredicates():
```

```
                self.sentence_map[predicate] = self.sentence_map.get(predicate, []) + [sentence]
```

```
    def convertSentencesToCNF(self):
```

```
        for sentenceldx in range(len(self.inputSentences)):
```

```
            if "=>" in self.inputSentences[sentenceldx]: # Do negation of the Premise and add them as  
literal
```

```
                self.inputSentences[sentenceldx] = negateAntecedent(self.inputSentences[sentenceldx])
```

```

def askQueries(self, queryList):
    results = []

    for query in queryList:
        negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
        negatedPredicate = negatedQuery.predicates[0]
        prev_sentence_map = copy.deepcopy(self.sentence_map)

        self.sentence_map[negatedPredicate.name] = self.sentence_map.get(negatedPredicate.name,
[]) + [negatedQuery]

        self.timeLimit = time.time() + 40

        try:
            result = self.resolve([negatedPredicate], [False]*(len(self.inputSentences) + 1))
        except:
            result = False

        self.sentence_map = prev_sentence_map

        if result:
            results.append("TRUE")
        else:
            results.append("FALSE")

    return results

def resolve(self, queryStack, visited, depth=0):
    if time.time() > self.timeLimit:
        raise Exception

    if queryStack:
        query = queryStack.pop(-1)
        negatedQuery = query.getNegatedPredicate()

```

```

queryPredicateName = negatedQuery.name
if queryPredicateName not in self.sentence_map:
    return False
else:
    queryPredicate = negatedQuery
    for kb_sentence in self.sentence_map[queryPredicateName]:
        if not visited[kb_sentence.sentence_index]:
            for kbPredicate in kb_sentence.findPredicates(queryPredicateName):

                canUnify, substitution = performUnification(copy.deepcopy(queryPredicate),
copy.deepcopy(kbPredicate))

                if canUnify:
                    newSentence = copy.deepcopy(kb_sentence)
                    newSentence.removePredicate(kbPredicate)
                    newQueryStack = copy.deepcopy(queryStack)

                    if substitution:
                        for old, new in substitution.items():
                            if old in newSentence.variable_map:
                                parameter = newSentence.variable_map[old]
                                newSentence.variable_map.pop(old)
                                parameter.unify("Variable" if new[0].islower() else "Constant", new)
                                newSentence.variable_map[new] = parameter

                    for predicate in newQueryStack:
                        for index, param in enumerate(predicate.params):
                            if param.name in substitution:
                                new = substitution[param.name]
                                predicate.params[index].unify("Variable" if new[0].islower() else
"Constant", new)

```



```

        for predicate in newSentence.predicates:
            newQueryStack.append(predicate)

        new_visited = copy.deepcopy(visited)
        if kb_sentence.containsVariable() and len(kb_sentence.predicates) > 1:
            new_visited[kb_sentence.sentence_index] = True

        if self.resolve(newQueryStack, new_visited, depth + 1):
            return True

    return False

return True

```

```

def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
                    return False, {}
        else:
            return False, {}
    else:

```

```

    if not query.isConstant():
        if kb.name not in substitution:
            substitution[kb.name] = query.name
        elif substitution[kb.name] != query.name:
            return False, {}
        kb.unify("Variable", query.name)
    else:
        if kb.name not in substitution:
            substitution[kb.name] = query.name
        elif substitution[kb.name] != query.name:
            return False, {}
    return True, substitution

```

```

def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate

```

```

def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)

```

```

def getInput(filename):
    with open(filename, "r") as file:

```

```

noOfQueries = int(file.readline().strip())
inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
noOfSentences = int(file.readline().strip())
inputSentences = [file.readline().strip() for _ in range(noOfSentences)]
return inputQueries, inputSentences

```

```

def printOutput(filename, results):

```

```

    print(results)

```

```

    with open(filename, "w") as file:

```

```

        for line in results:

```

```

            file.write(line)

```

```

            file.write("\n")

```

```

    file.close()

```

```

if __name__ == '__main__':

```

```

    inputQueries_, inputSentences_ = getInput("RA1911003010305/input.txt")

```

```

    knowledgeBase = KB(inputSentences_)

```

```

    knowledgeBase.prepareKB()

```

```

    results_ = knowledgeBase.askQueries(inputQueries_)

```

```

    printOutput("RA1911003010305/output.txt", results_)

```

Input

2

Friends(Alice,Bob,Charlie,Diana)

Friends(Diana,Charlie,Bob,Alice)

2

Friends(a,b,c,d)

NotFriends(a,b,c,d)

Output

```
253     knowledgeBase = kb(inputSentences_)
254     knowledgeBase.prepareKB()
255     results_ = knowledgeBase.askQueries(inputQueries_)
256     printOutput("RA1911003010305/output.txt", results_)
```

Immediate × RA191100301030 × RA191100301030 × RA191100301030 × RA191100301030 × RA191100301030 ×

Run Command: RA1911003010305/resolution.py

```
['TRUE', 'TRUE']
```

Process exited with code: 0

Result: Unification of expression was done and the conversion set was printed and the result of all queries in input file were printed and written to output.txt .

EXP-8

Date:06/04/2022

Implementation of a learning algorithm – linear regression

Working Principle:

Linear regression shows the linear relationship between the independent variable (X-axis) and the dependent variable (Y-axis). To calculate best-fit line linear regression uses a traditional slope-intercept form. A regression line can be a Positive Linear Relationship or a Negative Linear Relationship. The goal of the linear regression algorithm is to get the best values for a_0 and a_1 to find the best fit line and the best fit line should have the least error. In Linear Regression, Mean Squared Error (MSE) cost function is used, which helps to figure out the best possible values for a_0 and a_1 , which provides the best fit line for the data points. Using the MSE function, we will change the values of a_0 and a_1 such that the MSE value settles at the minima. Gradient descent is a method of updating a_0 and a_1 to minimize the cost function (MSE).

Source code:

```
import numpy as np

from sklearn.linear_model import LinearRegression

x = np.array([5, 15, 25, 35, 45, 55]).reshape((-1, 1))

y = np.array([5, 20, 14, 32, 22, 38])

model = LinearRegression()

model.fit(x, y)
```

```

r_sq = model.score(x, y)

print('coefficient of determination:', r_sq)

print('intercept:', model.intercept_)

print('slope:', model.coef_) new_model = LinearRegression().fit(x, y.reshape((-1, 1)))
print('intercept:', new_model.intercept_)

intercept: [5.63333333]

print('slope:', new_model.coef_)

y_pred = model.predict(x)

print('predicted response:', y_pred, sep='\n')

y_pred = model.intercept_ + model.coef_ * x

print('predicted response:', y_pred, sep='\n') x_new = np.arange(5).reshape((-1, 1))

print(x_new)

y_new = model.predict(x_new)

print(y_new)

```

OutPut

```

x_new = np.arange(5).reshape((-1, 1))
print(x_new)
y_new = model.predict(x_new)
print(y_new)

coefficient of determination: 0.715875613747954
intercept: 5.633333333333333
slope: [0.54]
intercept: [5.63333333]
slope: [[0.54]]
predicted response:
[ 8.33333333 13.73333333 19.13333333 24.53333333 29.93333333 35.33333333]
predicted response:
[[ 8.33333333]
 [13.73333333]
 [19.13333333]
 [24.53333333]
 [29.93333333]
 [35.33333333]]
[[0]
 [1]
 [2]
 [3]
 [4]]
[5.63333333 6.17333333 6.71333333 7.25333333 7.79333333]

```

Result:

Hence, the Implementation of Linear Regression as a machine learning algorithm is done successfully

EXPERIMENT NO: 10

IMPLEMENTATION OF NLP – TAGGING & PARTS OF SPEECH

Working Principle:

In natural language processing, human language is separated into fragments so that the grammatical structure of sentences and the meaning of words can be analyzed and understood in context.

- **Part -of-speech -tagging** : marking up words as nouns, verbs, adjective s, adverbs, pronouns, etc

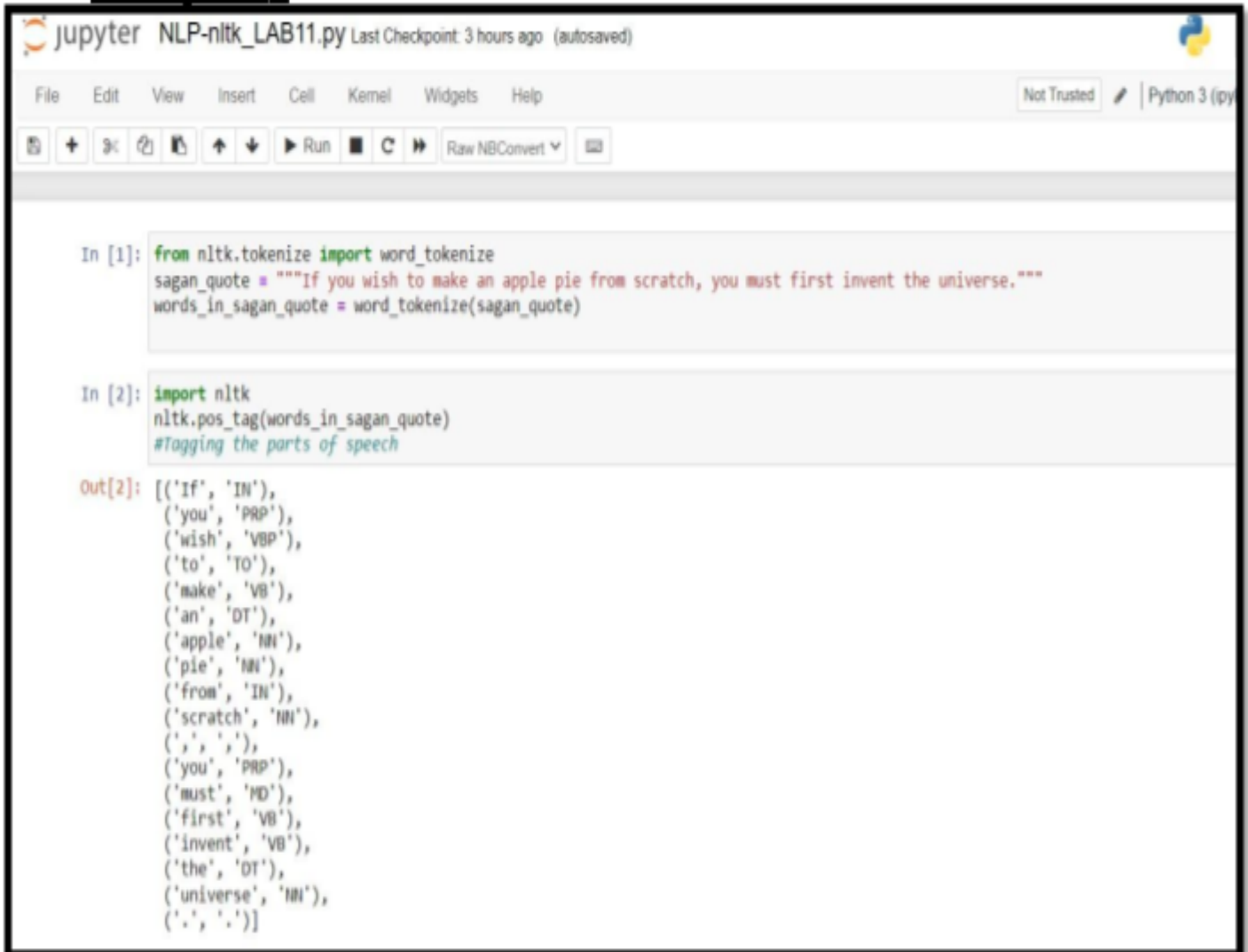
In python the availability of nltk makes the working of nlp very easy and efficient.

The word tokeniser splits the given sentence into words and then the pos_tag helps in identification of the the parts of speech and tag them accordingly.

Source code:

```
from nltk.tokenize import word_tokenize
sagan_quote = """If you wish to make an apple pie from scratch, you must first
invent the universe."""
words_in_sagan_quote = word_tokenize(sagan_quote)
import nltk
nltk.pos_tag(words_in_sagan_quote)
#Tagging the parts of speech
```

Output:



The image shows a Jupyter Notebook interface with the title 'NLP-nltk_LAB11.py'. The notebook contains two input cells and one output cell. The first input cell imports the 'word_tokenize' function from 'nltk.tokenize' and tokenizes a quote by Sagan. The second input cell imports 'nltk' and uses 'pos_tag' to tag the tokens. The output cell displays the resulting list of word-tag pairs.

```
jupyter NLP-nltk_LAB11.py Last Checkpoint: 3 hours ago (autosaved)
```

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 (ipy)

In [1]: `from nltk.tokenize import word_tokenize`
`sagan_quote = """If you wish to make an apple pie from scratch, you must first invent the universe."""`
`words_in_sagan_quote = word_tokenize(sagan_quote)`

In [2]: `import nltk`
`nltk.pos_tag(words_in_sagan_quote)`
#Tagging the parts of speech

Out[2]: `[('If', 'IN'), ('you', 'PRP'), ('wish', 'VBP'), ('to', 'TO'), ('make', 'VB'), ('an', 'DT'), ('apple', 'NN'), ('pie', 'NN'), ('from', 'IN'), ('scratch', 'NN'), (',', ','), ('you', 'PRP'), ('must', 'MD'), ('first', 'VB'), ('invent', 'VB'), ('the', 'DT'), ('universe', 'NN'), ('.', '.')]`

Result:

Hence, the Implementation of NLP for tagging parts of speech is done successfully.

EXP No:10

IMPLEMENTATION OF DEEP LEARNING - KERAS-MODEL

Working Principle:

Keras is a deep learning algorithm tool that wraps the efficient numerical computation libraries Theano and TensorFlow and allows you to define and train neural network models in just a few lines of code.

The steps to be followed are:

1. Load Data.
2. Define Keras Model.
3. Compile Keras Model.
4. Fit Keras Model.
5. Evaluate Keras Model.
6. Tie It All Together.
7. Make Predictions

Source code:

```
# first neural network with keras make predictions
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10, verbose=0)
# evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f % (accuracy*100))
# make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)
# summarize the first 5 cases
for i in range(5):
    print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

Output:

```
jupyter DL-KERAS_MODEL_LAB12 Last Checkpoint: 3 hours ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

In [1]: # first neural network with keras make predictions
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10, verbose=0)
# evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f % (accuracy*100))
# make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)
# summarize the first 5 cases
for i in range(5):
    print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))

24/24 [*****] - 0s 1ms/step - loss: 0.4583 - accuracy: 0.7891
Accuracy: 78.91
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] => 1 (expected 1)
[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] => 0 (expected 0)
[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] => 1 (expected 1)
[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] => 0 (expected 0)
[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] => 1 (expected 1)
```

Result:

Hence, the Implementation of Deep Learning for Keras Model is done successfully.