**SEARCH ENGINE**

**17BCE0398**

#include<iostream>

#include<conio.h>

#include<stdlib.h>

#include<time.h>

using namespace std;

int cost[10][10],i,j,k,n,stk[10],top,v,visit[10],visited[10];

main()

{

int m;

cout <<"enterno of vertices";

cin >> n;

cout <<"ente no of edges";

cin >> m;

cout <<"\nEDGES \n";

for(k=1;k<=m;k++)

{

cin >>i>>j;

cost[i][j]=1;

}

cout <<"enter initial vertex";

cin >>v;

cout <<"ORDER OF VISITED VERTICES";

cout << v <<" ";

clock\_t tStart = clock();

visited[v]=1;

k=1;

while(k<n)

{

for(j=n;j>=1;j--)

if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)

{

visit[j]=1;

stk[top]=j;

top++;

}

v=stk[--top];

cout<<v << " ";

k++;

visit[v]=0; visited[v]=1;

}

cout<<"Time taken:\n";

cout<<(double)(clock() - tStart)/CLOCKS\_PER\_SEC;

}

#include <iostream>

#include <vector>

#include<time.h>

using namespace std;

// Data structure to store graph edges

struct Edge {

int src, dest;

};

// represent a graph object

class Graph

{

public:

// An array of vectors to represent adjacency list

vector<int> \*adjList;

// Constructor

Graph(vector<Edge> const &edges, int N)

{

// allocate memory

adjList = new vector<int>[N];

// add edges to the Directed graph

for (unsigned i = 0; i < edges.size(); i++)

{

int src = edges[i].src;

int dest = edges[i].dest;

adjList[src].push\_back(dest);

}

}

~Graph() {

delete[] adjList;

}

};

// Perform DFS on graph and set departure time of all

// vertices of the graph

int DFS(Graph const &graph, int v, vector<bool>

&discovered, vector<int> &departure, int& time)

{

// mark current node as discovered

discovered[v] = true;

// set arrival time

time++;

// doing for every edge (v -> u)

for (int u : graph.adjList[v])

{

// u is not discovered

if (!discovered[u])

DFS(graph, u, discovered, departure, time);

}

// ready to backtrack

// set departure time of vertex v

departure[time] = v;

time++;

}

// performs on a given DAG

void donewSearch(Graph const& graph, int N)

{

// departure stores vertex number having its departure

// time equal to the index of it

vector<int> departure(2\*N, -1);

// Note if i had done the other way around i.e. fill

// array with departure time by using vertex number

// as index, we would need to sort the array later

// stores vertex is discovered or not

vector<bool> discovered(N);

int time = 0;

// perform DFS on all undiscovered vertices

for (int i = 0; i < N; i++)

if (!discovered[i])

DFS(graph, i, discovered, departure, time);

// Print the vertices in order of their decreasing

// departure time in DFS

for (int i = 2\*N - 1; i >= 0; i--)

if (departure[i] != -1)

cout << departure[i] << " ";

}

// newsearch Algorithm for a DAG using DFS

int main()

{

// vector of graph edges as per above diagram

clock\_t tStart = clock();

vector<Edge> edges =

{

{0, 6}, {1, 2}, {1, 4}, {1, 6}, {3, 0}, {3, 4},

{5, 1}, {7, 0}, {7, 1}

};

// Number of nodes in the graph

int N = 9;

// create a graph from edges

Graph graph(edges, N);

// perform newsearch

donewSearch(graph, N);

cout<<"Time taken:\n";

cout<<(double)(clock() - tStart)/CLOCKS\_PER\_SEC;

return 0;

}

**17BCE0993**

# import

import numpy as np

import scipy as sc

import pandas as pd

from fractions import Fraction

# keep it tidy

def float\_format(vector, decimal):

return np.round((vector).astype(np.float), decimals=decimal)

# we have 3 webpages and probability of landing to each one is 1/3

#(defaultProbability)

dp = Fraction(1,3)

# matrix

M = np.matrix([[0,0,1],

[Fraction(1,2),0,0],

[Fraction(1,2),1,0]])

E = np.zeros((3,3))

E[:] = dp

# damping factor

beta = 0.7

# matrix

A = beta \* M + ((1-beta) \* E)

# initial vector

r = np.matrix([dp, dp, dp])

r = np.transpose(r)

previous\_r = r

for it in range(1,100):

r = A \* r

print(float\_format(r,3))

#check if converged

if (previous\_r==r).all():

break

previous\_r = r

print ("Final:\n", float\_format(r,3))

print ("sum", np.sum(r))