**ARYAMAN MISHRA**

**19BCE1027**

**AIM:**

Input a directed graph (web pages as nodes and links between pages as edges), damping factor and number of iterations as part of implementing the following algorithms:

1. Page rank
2. Weighted page rank

to output the ranking of web pages.

**DATA STRUCTURES**:DICTIONARIES,2D ARRAYS,GRAPHS

**ALGORITHM:**

**1.Page Rank**

Return the PageRank of the nodes in the graph.

    PageRank computes a ranking of the nodes in the graph G based on

    the structure of the incoming links. It was originally designed as

    an algorithm to rank web pages.

    Parameters

    ----------

    G : graph

      A NetworkX graph.  Undirected graphs will be converted to a directed

      graph with two directed edges for each undirected edge.

    alpha : float, optional

      Damping parameter for PageRank, default=0.85.

    personalization: dict, optional

      The "personalization vector" consisting of a dictionary with a

      key for every graph node and nonzero personalization value for each node.

      By default, a uniform distribution is used.

    max\_iter : integer, optional

      Maximum number of iterations in power method eigenvalue solver.

    tol : float, optional

      Error tolerance used to check convergence in power method solver.

    nstart : dictionary, optional

      Starting value of PageRank iteration for each node.

    weight : key, optional

      Edge data key to use as weight.  If None weights are set to 1.

    dangling: dict, optional

      The outedges to be assigned to any "dangling" nodes, i.e., nodes without

      any outedges. The dict key is the node the outedge points to and the dict

      value is the weight of that outedge. By default, dangling nodes are given

      outedges according to the personalization vector (uniform if not

      specified). This must be selected to result in an irreducible transition

      matrix (see notes under google\_matrix). It may be common to have the

      dangling dict to be the same as the personalization dict.

    Returns

    -------

    pagerank : dictionary

       Dictionary of nodes with PageRank as value

    Notes

    -----

    The eigenvector calculation is done by the power iteration method

    and has no guarantee of convergence.  The iteration will stop

    after max\_iter iterations or an error tolerance of

    number\_of\_nodes(G)\*tol has been reached.

    The PageRank algorithm was designed for directed graphs but this

    algorithm does not check if the input graph is directed and will

    execute on undirected graphs by converting each edge in the

    directed graph to two edges.

**2.Weighted page rank**

Weighted PageRank algorithm assigns higher rank values to more popular (important) pages instead of dividing the rank value of a page evenly among its outlink pages. Each outlink page gets a value proportional to its popularity, i.e. its number of inlinks and outlinks.

To a webpage ‘u’, an inlink is a URL of another webpage which contains a link pointing to ‘u’. Similarly to webpage ‘u’, an outlink is a link appearing in ‘u’ which points to another webpage. The number of inlinks is represented by **Win(v,u)**and the number of outlinks is represented as **Wout(v,u)**.

**Win(v,u)**is the weight of link (v, u) calculated based on the number of inlinks of page u and the number of inlinks of all reference pages of page v.



Here, Ipand Iurepresent the number of inlinks of page ‘p’ and ‘u’ respectively. R(v)represents the list of all reference pages of page ‘v’.

**Wout(v,u)**is the weight of link (v, u) calculated based on the number of outlinks of page u and the number of outlinks of all reference pages of page v.



Here, Op and Ou represent the number of outlinks of page ‘p’ and ‘u’ respectively. R(v) represents the list of all reference pages of page ‘v’.

Based on the importance of all pages as describes by their number of inlinks and outlinks, the Weighted PageRank formula is given as:



Here, **PR(x)** refers to the Weighted PageRank of page x.

**IMPLEMENTATION CODE AND RESULTS:**

**1.WEIGHTED PAGE RANK**

def win(matrix, m, o):

    k = 0

    for i in range(0, n):

        if(int(matrix[i][m]) == 1):

            k = k+1

    l = 0

    for i in range(0, n):

        if(int(matrix[o][i] == 1)):

            for j in range(0, n):

                if(matrix[j][i] == 1):

                    l = l+1

    return float(k/l)

def wout(matrix, m, o):

    k = 0

    for i in range(0, n):

        if(int(matrix[0][i]) == 1):

            k = k+1

    l = 0

    for i in range(0, n):

        if(int(matrix[o][i] == 1)):

            for j in range(0, n):

                if(matrix[i][j] == 1):

                    l = l+1

    return float(k/l)

def pagerank(matrix, o, n, p):

    a = 0

    for i in range(0, n):

        if(int(matrix[i][o]) == 1):

            k = 0

            for s in range(0, n):

                if(matrix[i][s] == 1):

                    k = k+1

            a = a+float((p[i]/k)\*win(matrix, i, o)\*wout(matrix, i, o))

    return a

#n = 5

#matrix = [[0, 1, 1, 1, 0], [1, 0, 1, 1, 0], [0, 0, 0, 1, 0], [0, 0, 1, 0, 1], [0, 1, 1, 1, 0]]

#d = 0.25  # damping factor

#o = 5

print("Enter number of web pages(nodes).")

n=int(input())

matrix = []

print("Enter the links between edges in 0 and 1:")

for i in range(n):

    a =[]

    for j in range(n):

         a.append(int(input()))

    matrix.append(a)

print("Enter the damping factor:")

d=float(input())

print("Enter the number of iterations:")

o=int(input())

print("Number of iterations is:", o)

sum = 0

p = []

print("Page    Iteration   Page Rank")

for i in range(0, n):

    p.append(1)

for k in range(0, o):

    for u in range(0, n):

        g = pagerank(matrix, u, n, p)

        p[u] = (1-d)+d\*g

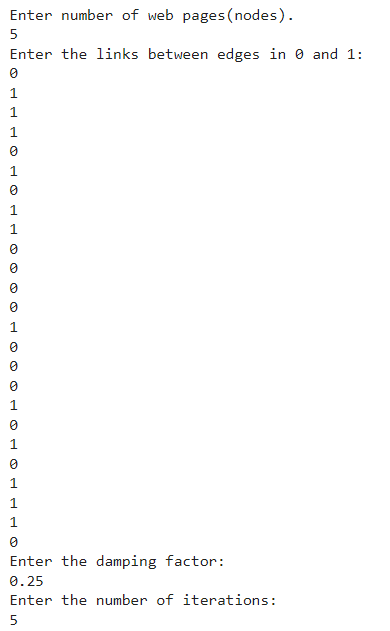
        print("Page ",u+1," ",k+1," ",p[u])

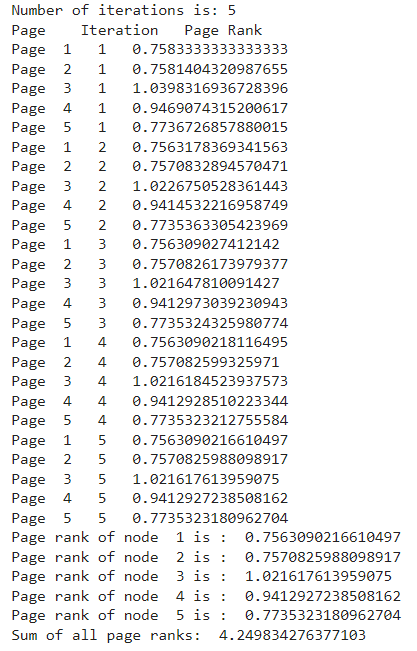
for i in range(0, n):

    sum += p[i]

    print("Page rank of node ", i+1, "is : ", p[i])

print("Sum of all page ranks: ", sum)





**2.Page Rank**

def pagerank(G, alpha=0.85, personalization=None,

      max\_iter=i, tol=1.0e-6, nstart=None, weight='weight',

      dangling=None):

  if len(G) == 0:

    return {}

  if not G.is\_directed():

    D = G.to\_directed()

  else:

    D = G

  # Create a copy in (right) stochastic form

  W = nx.stochastic\_graph(D, weight=weight)

  N = W.number\_of\_nodes()

  # Choose fixed starting vector if not given

  if nstart is None:

    x = dict.fromkeys(W, 1.0 / N)

  else:

    # Normalized nstart vector

    s = float(sum(nstart.values()))

    x = dict((k, v / s) for k, v in nstart.items())

  if personalization is None:

    # Assign uniform personalization vector if not given

    p = dict.fromkeys(W, 1.0 / N)

  else:

    missing = set(G) - set(personalization)

    if missing:

      raise NetworkXError('Personalization dictionary '

                'must have a value for every node. '

                'Missing nodes %s' % missing)

    s = float(sum(personalization.values()))

    p = dict((k, v / s) for k, v in personalization.items())

  if dangling is None:

    # Use personalization vector if dangling vector not specified

    dangling\_weights = p

  else:

    missing = set(G) - set(dangling)

    if missing:

      raise NetworkXError('Dangling node dictionary '

                'must have a value for every node. '

                'Missing nodes %s' % missing)

    s = float(sum(dangling.values()))

    dangling\_weights = dict((k, v/s) for k, v in dangling.items())

  dangling\_nodes = [n for n in W if W.out\_degree(n, weight=weight) == 0.0]

  # power iteration: make up to max\_iter iterations

  for \_ in range(max\_iter):

    xlast = x

    x = dict.fromkeys(xlast.keys(), 0)

    danglesum = alpha \* sum(xlast[n] for n in dangling\_nodes)

    for n in x:

      # this matrix multiply looks odd because it is

      # doing a left multiply x^T=xlast^T\*W

      for nbr in W[n]:

        x[nbr] += alpha \* xlast[n] \* W[n][nbr][weight]

      x[n] += danglesum \* dangling\_weights[n] + (1.0 - alpha) \* p[n]

    # check convergence, l1 norm

    err = sum([abs(x[n] - xlast[n]) for n in x])

    if err < N\*tol:

      return x

  raise NetworkXError('pagerank: power iteration failed to converge '

            'in %d iterations.' % max\_iter)

import networkx as nx

print("Enter nodes,edges,damping factor and number of iterations")

n=int(input())

e=int(input())

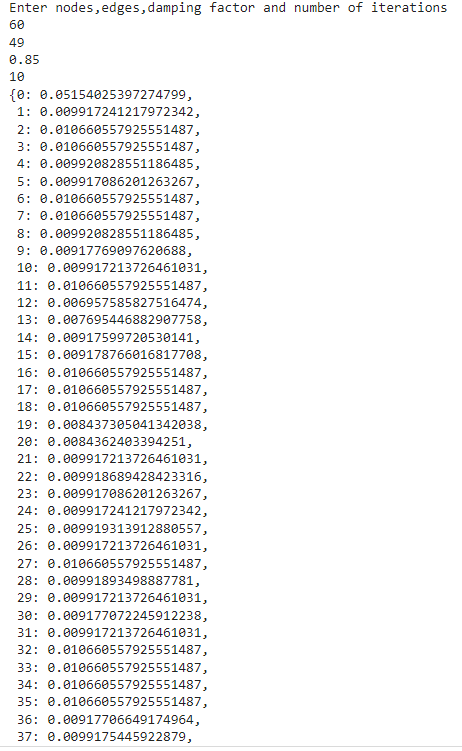
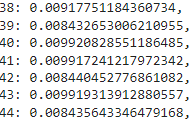
d=float(input())

i=int(input())

G=nx.barabasi\_albert\_graph(n,e)

pr=nx.pagerank(G,d)

pr

import networkx as nx

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import operator

import random as rd

# created a directed graph

graph=nx.gnp\_random\_graph(25,0.6,directed=True)

#draw a graph

nx.draw(graph,with\_labels=True,font\_color='red',font\_size=10,node\_color='yellow')

#plot a graph

plt.show()

#number of nodes for graph

count=graph.number\_of\_nodes()

#graph neighbours of a node 1

print(list(graph.neighbors(1)))

#Page Rank Algorithm-Calculating random walk score

rank\_dict={}

x=rd.randint(0,25)

for j in range(0,25):

  rank\_dict[j]=0

rank\_dict[x]=rank\_dict[x]+1

for i in range(600000):

  list\_n=list(graph.neighbors(x))

  if(len(list\_n)==0):

    x=rd.randint(0,25)

    rank\_dict[x]=rank\_dict[x]+1

  else:

    x=rd.choice(list\_n)

    rank\_dict[x]=rank\_dict[x]+1

print("Random Walk Score Updated")

#normalising values

for j in range(0,25):

  rank\_dict[j]=rank\_dict[j]/600000

#Page rank by networkx library

pagerank=nx.pagerank(graph)

#sorting both dictionaries based on items

pagerank\_sorted=sorted(pagerank.items(),key=lambda v:(v[1],v[0]),reverse=True)

pagerank\_sorted

#sorting the rank\_dict based on values

rank\_dict\_sorted=sorted(rank\_dict.items(),key=lambda v:(v[1],v[0]),reverse=True)

rank\_dict\_sorted

print("The order generated by our implementation algorithm is\n")

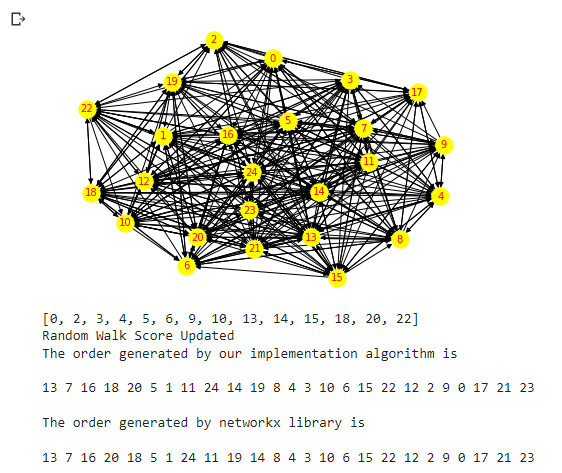
for i in rank\_dict\_sorted:

  print(i[0],end=" ")

print("\n\nThe order generated by networkx library is\n")

for i in pagerank\_sorted:

  print(i[0],end=" ")



CONCLUSION:ALL TASKS HAVE BEEN SUCCESFULLY IMPLEMENTED AND EXECUTED.