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 $\mathbf{W^{in}_{(v,u)}}$ and the number of outlinks is represented as $\mathbf{W^{out}_{(v,u)}}$.

$\mathbf{W^{in}_{(v,u)}}$ 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.

$$W^{in}(v,u) = I_u / \left(\sum_{p \in R(v)} I_p\right)$$

Here, I_p and I_u represent the number of inlinks of page 'p' and 'u' respectively.
R(v) represents the list of all reference pages of page 'v'.
$W^{out}_{(v,u)}$ 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.

$$W^{out}(v,u) = O_u / \left(\Sigma_{p \in R(v)} O_p\right)$$

Here, O_p and O_u 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:

$$PR(u) = (1-d) + d\sum_{v \in B(u)} PR(v) * W^{in}_{(v,u)} * W^{out}(v,u)$$

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)
```

```
Enter number of web pages(nodes).
5
Enter the links between edges in 0 and 1:
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
Enter the damping factor:
0.25
Enter the number of iterations:
5
```

```
Number of iterations is: 5
Page    Iteration   Page Rank
Page  1   1   0.7583333333333333
Page  2   1   0.7581404320987655
Page  3   1   1.0398316936728396
Page  4   1   0.9469074315200617
Page  5   1   0.7736726857880015
Page  1   2   0.7563178369341563
Page  2   2   0.7570832894570471
Page  3   2   1.0226750528361443
Page  4   2   0.9414532216958749
Page  5   2   0.7735363305423969
Page  1   3   0.756309027412142
Page  2   3   0.7570826173979377
Page  3   3   1.021647810091427
Page  4   3   0.9412973039230943
Page  5   3   0.7735324325980774
Page  1   4   0.7563090218116495
Page  2   4   0.757082599325971
Page  3   4   1.0216184523937573
Page  4   4   0.9412928510223344
Page  5   4   0.7735323212755584
Page  1   5   0.7563090216610497
Page  2   5   0.7570825988098917
Page  3   5   1.021617613959075
Page  4   5   0.9412927238508162
Page  5   5   0.7735323180962704
Page rank of node  1 is :  0.7563090216610497
Page rank of node  2 is :  0.7570825988098917
Page rank of node  3 is :  1.021617613959075
Page rank of node  4 is :  0.9412927238508162
Page rank of node  5 is :  0.7735323180962704
Sum of all page ranks:  4.249834276377103
```

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:
```

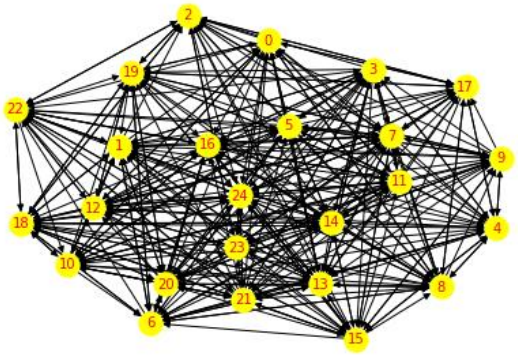
```
  D = G

# 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
```

```
Enter nodes,edges,damping factor and number of iterations
60
49
0.85
10
{0: 0.05154025397274799,
 1: 0.009917241217972342,
 2: 0.010660557925551487,
 3: 0.010660557925551487,
 4: 0.009920828551186485,
 5: 0.009917086201263267,
 6: 0.010660557925551487,
 7: 0.010660557925551487,
 8: 0.009920828551186485,
 9: 0.00917769097620688,
 10: 0.009917213726461031,
 11: 0.010660557925551487,
 12: 0.006957585827516474,
 13: 0.007695446882907758,
 14: 0.00917599720530141,
 15: 0.009178766016817708,
 16: 0.010660557925551487,
 17: 0.010660557925551487,
 18: 0.010660557925551487,
 19: 0.008437305041342038,
 20: 0.0084362403394251,
 21: 0.009917213726461031,
 22: 0.009918689428423316,
 23: 0.009917086201263267,
 24: 0.009917241217972342,
 25: 0.009919313912880557,
 26: 0.009917213726461031,
 27: 0.010660557925551487,
 28: 0.00991893498887781,
 29: 0.009917213726461031,
 30: 0.009917072245912238,
 31: 0.009917213726461031,
 32: 0.010660557925551487,
 33: 0.010660557925551487,
 34: 0.010660557925551487,
 35: 0.010660557925551487,
 36: 0.00917706649174964,
 37: 0.0099175445922879,
```

```
38: 0.00917751184360734,
39: 0.008432653006210955,
40: 0.009920828551186485,
41: 0.009917241217972342,
42: 0.0084404527768610 82,
43: 0.009919313912880557,
44: 0.008435643346479168,
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='y
ellow')

#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=T
rue)
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=" ")
```




```
[0, 2, 3, 4, 5, 6, 9, 10, 13, 14, 15, 18, 20, 22]
Random Walk Score Updated
The order generated by our implementation algorithm is

13 7 16 18 20 5 1 11 24 14 19 8 4 3 10 6 15 22 12 2 9 0 17 21 23

The order generated by networkx library is

13 7 16 20 18 5 1 24 11 19 14 8 4 3 10 6 15 22 12 2 9 0 17 21 23
```

CONCLUSION:ALL TASKS HAVE BEEN SUCCESFULLY IMPLEMENTED AND EXECUTED.