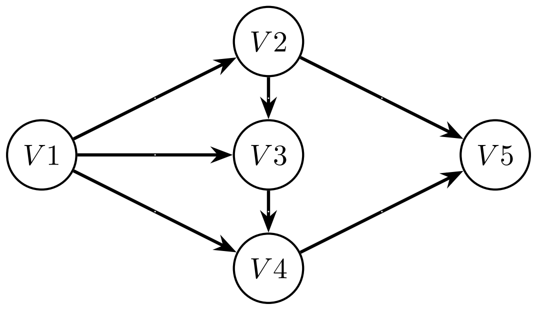
Tufts University - Department of Mathematics
Math 87 Homework Midterm 2 Solutions

Note that this is only a set of partial solutions and is not intended to represent a fully complete write up for the midterm. To receive full points on format, you should include details to explain each of the problems. Someone who is unfamiliar with the assignment should be able to read your report and have a clear understanding of the goal and results.

The problem introduces an example of a set of web pages linked as pictured below.



The corresponding adjacency matrix, A, and transition matrix, T, are given by:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}, \qquad T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 1 & 0 \end{bmatrix}.$$

Note that the transition matrix T is not a stochastic matrix (columns do not sum to one) since $V5$ is a sink. To modify this, we consider the case of a random-surfer who will randomly choose a site from all possible sites. This gives new transition matrix, T_1, which is a stochastic matrix:

$$T_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & \frac{1}{5} \\ \frac{1}{3} & 0 & 0 & 0 & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 & \frac{1}{5} \\ \frac{1}{3} & 0 & 1 & 0 & \frac{1}{5} \\ 0 & \frac{1}{2} & 0 & 1 & \frac{1}{5} \end{bmatrix}.$$

Next, a damping factor p is introduced, where p is the probability that the surfer will stop clicking, and in the case that they do stop clicking, there is equal probability that the stop node will be chosen at random from all available nodes. Therefore, the `PageRank` transition matrix is modeled by:

$$C = (1-p)T_1 + \frac{p}{5}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

1. Given an adjacency matrix A, we can determine the entries of the `PageRank` transition matrix using the description below.

(a) If the jth column of A is equal to $\mathbf{0}$, then for $i = 0, ..., n-1$: $C_{i,j} = \frac{1}{n}$.

If the adjacency matrix has a column of zeros (sink node), replace that column with $\frac{1}{n}\mathbb{1}_n^T$, where $\mathbb{1}_n$ is a row vector of all ones of length n. This means that each entry gets assigned value $\frac{1}{n}$. Then, following the definition of C, each entry is multiplied by $1-p$, and then $\frac{p}{n}$ is added. This gives $\frac{1-p}{n} + \frac{p}{n} = \frac{1}{n}$.

(b) Suppose the sum of the entries in the jth column of A is equal to $s > 0$. Then for each $i = 0, ..., n-1$: if $A_{i,j} = 0$, then $C_{i,j} = \frac{p}{n}$. If $A_{i,j} = 1$, then $C_{i,j} = \frac{1-p}{s} + \frac{p}{n}$.

If $A_{i,j} = 0$, then $(T_1)_{i,j} = 0$ and $(1-p)(T_1)_{i,j} = 0$, leaving only $C_{i,j} = \frac{p}{n}$. If $A_{i,j} = 1$, then $(T_1)_{i,j} = \frac{1}{s}$, and $C_{i,j} = \frac{1-p}{s} + \frac{p}{n}$.

2. Use your description in (1) to write a `python` function to create the `PageRank` transition matrix from an adjacency matrix A.

PageRank Transition Matrix

```
import numpy as np
def make_transition(A,p):
    (n,m) = A.shape
    if n==m:
        C = np.zeros((n,n)) ## create a matrix of
                            ## zeros of the correct size

        for j in range(m):
            s=A[:,j].sum()
            if s==0:
                C[:,j]=np.ones(n)/n
            else:
                C[:,j]=(1-p)*A[:,j]/s + p*np.ones(n)/n
        return C
```

Then, test this on the suggested adjacency matrix:

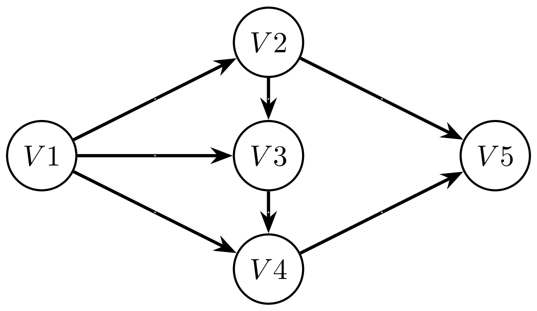
Test transition matrix function

```
float_formatter = "{:.5f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})
A = np.array([[ 0, 0, 0, 0, 0 ],
              [ 1, 0, 0, 0, 0 ],
              [ 1, 1, 0, 0, 0 ],
              [ 1, 0, 1, 0, 0 ],
              [ 0, 1, 0, 1, 0 ]])
make_transition(A,p=0.8)
```

```
array([[0.16000, 0.16000, 0.16000, 0.16000, 0.20000],
       [0.22667, 0.16000, 0.16000, 0.16000, 0.20000],
       [0.22667, 0.26000, 0.16000, 0.16000, 0.20000],
       [0.22667, 0.16000, 0.36000, 0.16000, 0.20000],
       [0.16000, 0.26000, 0.16000, 0.36000, 0.20000]])
```

The output is exactly what we expected.

3. Find a 1-eigenvector for the `PageRank` transition matrix C when $p = .8$ and when $p = .4$ for the directed graph:



Be sure to normalize your eigenvector to obtain probability vectors. Explain what the entries in this vector predict about the probability of a random surfer landing on one of the five web-sites corresponding to the nodes in the diagram. Compare the information you get from the eigenvector with that obtained by studying powers of C calculated using `np.linalg.matrix_power(C,m)`.

To compute C, call the function written in the previous problem with the adjacency matrix previously described. The eigenvalues and eigenvectors can be computed using `np.linalg.eig`. In order to make the computed eigenvector a probability vector, divide the vector by the sum of the entries. Then, compute C^m for $m = 2, 5, 10, 50$, and display the first column. We expect that as m gets large enough, the first column should converge to the eigenvector corresponding to $\lambda = 1$.

p = 0.8

```
C1=make_transition(A,p=0.8)
l1, v1= np.linalg.eig(C1)
print('First eigenvalue:', np.real(l1[0]))
w1=np.real(v1[:,0])
w1/=w1.sum()
print('Corresponding eigenvector:', w1)

print('First column of powers of C:')

M=[2,5,10,50]
for i in range(len(M)):
    print(' m='+str(M[i])+':',
          np.linalg.matrix_power(C1,M[i])[:,0])
```

```
First eigenvalue: 0.9999999999999989
Corresponding eigenvector: [0.16925 0.18054 0.19859 0.22026 0.23136]
First column of powers of C:
 m=2: [0.16640 0.17707 0.19973 0.22240 0.23440]
 m=5: [0.16926 0.18054 0.19860 0.22026 0.23135]
 m=10: [0.16925 0.18054 0.19859 0.22026 0.23136]
 m=50: [0.16925 0.18054 0.19859 0.22026 0.23136]
```

p = 0.4

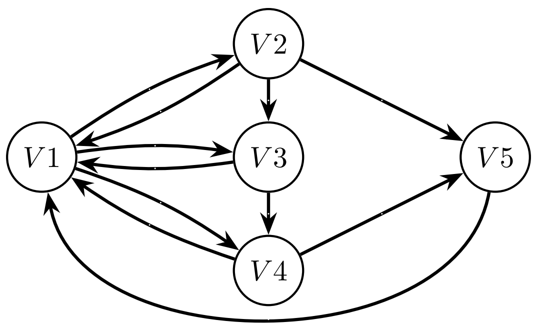
```
C2=make_transition(A,p=0.4)
l2, v2= np.linalg.eig(C2)
print('First eigenvalue:', np.real(l2[0]))
w2=np.real(v2[:,0])
w2/=w2.sum()
print('Corresponding eigenvector:', w2)

print('First column of powers of C:')

M=[2,5,10,50]
for i in range(len(M)):
    print(' m='+str(M[i])+':',
          np.linalg.matrix_power(C2,M[i])[:,0])
```

```
First eigenvalue: 1.0000000000000004
Corresponding eigenvector: [0.11713 0.14055 0.18272 0.25019 0.30941]
First column of powers of C:
 m=2: [0.08960 0.10560 0.18960 0.27360 0.34160]
 m=5: [0.11736 0.14096 0.18363 0.25129 0.30677]
 m=10: [0.11713 0.14055 0.18272 0.25019 0.30941]
 m=50: [0.11713 0.14055 0.18272 0.25019 0.30941]
```

In both cases, we see exactly what we would expect, which is as m gets larger, the first column of C converges to the 1-eigenvector. For $m = 2$ the first column of C^2 is not that close to the eigenvector. It's closer for $m = 5$, and for $m = 10$ and $m = 50$, they are the same to at least 6 decimal places.

The 1-eigenvector describes the long-term behavior of the system, meaning if the random surfer keeps clicking, which pages are they most likely to visit. While the probabilities are slightly different for $p = 0.8$ and $p = 0.4$, the results are similar. $V5$ is the most likely to be clicked, followed closely by $V4$. Then, $V3, V2, V1$ are all very close together in chances of being visited. This makes sense with the transition graph, as $V5$ is a sink in the original diagram, and $V4$ has 3 links to it, making them mostly likely to be visited. It also makes sense that the probabilities for $V5$ and $V4$ are higher for the lower damping factor since that lowers the chances of navigating to any page at random, meaning the system behaves more closely to the system described in the original graph.

4. The following diagram describes the same nodes as in problem 2., but includes some additional edges (i.e. links).



Assess the impact of the additional edges on the page rankings (when $p = .8$ and when $p = .4$) – i.e. compare the ranking obtained for this diagram with that obtained in problem 2. Before making your comparison, you'll need to first compute the adjacency matrix for this new configuration, and then use your code to find the corresponding `PageRank` transition matrix.

First, write down the new adjacency matrix.

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Next, use the function written in problem 2 to find the `PageRank` transition matrix, and again find the 1-eigenvector for $p = 0.8$ and $p = 0.4$.

New transition graph

```
#adjacency matrix
A2=np.empty((5,5))
A2[:,0]=[0, 1, 1, 1, 0]
A2[:,1]=[1, 0, 1, 0, 1]
A2[:,2]=[1, 0, 0, 1, 0]
A2[:,3]=[1, 0, 0, 0, 1]
A2[:,4]=[1, 0, 0, 0, 0]

#do eigen-decomp for p=0.8 and p=0.4
e3, v3 = np.linalg.eig(make_transition(A2,p=0.8))
print('p=0.8 eigenvector:', np.real(v3[:,0]/v3[:,0].sum()))
e4, v4 = np.linalg.eig(make_transition(A2,p=0.4))
print('p=0.4 eigenvector:', np.real(v4[:,0]/v4[:,0].sum()))
```

```
p=0.8 eigenvector: [0.24841 0.17656 0.18833 0.19539 0.19131]
p=0.4 eigenvector: [0.31990 0.14398 0.17277 0.19581 0.16754]
```

The long-term behavior is different for the two damping factors now. For $p = 0.8$, meaning we are likely to stop-clicking after only a few clicks, the probabilities of landing on $V2$ –$V5$ are now all very similar, and there's a slightly higher probability of clicking on $V1$. However, in the $p = 0.4$ case, we are less likely to stop clicking after a few clicks or navigate randomly, so $V1$ is the most likely to be clicked quite significantly. Since all new edges point toward $V1$, it makes sense that $V1$ is now the page to most likely be navigated to for both cases.

5. For any directed graph, explain why the corresponding `PageRank` transition matrix for $p > 0$ is a stochastic matrix corresponding to a strongly connected aperiodic transition diagram. In particular, explain why the conclusion of the Perron-Frobenius Theorem holds for this matrix.

   First, let's show that the `PageRank` matrix C is a stochastic matrix. Recall that C is defined as
   $$C = (1-p)T_1 + \frac{p}{n}\mathbb{1}_n^T\mathbb{1}_n,$$
   where $\mathbb{1}_n$ is a vector of length n of all ones, and $\mathbb{1}_n^T\mathbb{1}_n$ is an $n \times n$ matrix of all ones, and T_1 is a stochastic matrix. We want to show that s_j, the sum of the jth column of C, is equal to 1.

   $$\begin{aligned} s_j &= \sum_{i=1}^{n} C_{i,j} \\ &= \sum_{i=1}^{n}(1-p)\left(T_1\right)_{i,j} + \frac{p}{n}\left(\mathbb{1}_n^T\mathbb{1}_n\right)_{i,j} \\ &= (1-p)\sum_{i=1}^{n}\left(T_1\right)_{i,j} + \frac{p}{n}\sum_{i=1}^{n} 1 \\ &= (1-p)\cdot 1 + \frac{p}{n}\cdot n \\ &= 1 - p + p \\ &= 1. \end{aligned}$$

   Since T_1 is stochastic, the columns must sum to one. We also know that the column of the ones matrix sums to n. Multiplying by the matrix coefficients gives the desired result.

   The graph is strongly connected and aperiodic because of the $\frac{p}{n}\mathbb{1}_n^T\mathbb{1}_n$ component. As long as $p \neq 0$, this guarantees that every node is connected to every other node (strongly connected), and since every node connected to every other node, we can find cycles of arbitrary length in the graph (aperiodic). The addition of the damping factor guarantees that the Perron-Frobenius theorem is satisfied.

6. Extract the formatted data (`json`) from a file. Once you have produced the adjacency matrix A from `data.json`, use the `make_transition` function you wrote in the previous exercises to build the `PageRank` transition matrix C from the adjacency matrix A, for $p = 0.8$ Now find the page-rankings of these pages, using both the eigenvector method and the power-iteration method. For each method, report the top ten "sites" by name (when $p = 0.8$). When using the power-iteration method, report the number of iterations used. What happens if you reduce the value of p? How much must you vary p to see any change in the "top-ten" list?

Data extraction

```
import json
import numpy as np
float_formatter = "{:.5f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})
def bv(it,items):
    return np.array([1.0 if i == items.index(it)
        else 0.0 for i in range(len(items))])

def adj_from_json(json_file):
    with open(json_file) as f:
        adj_data = json.load(f)
    dict = {}
    for i in adj_data:
        lfrom = i['from']
        lto = i['to']
        if lfrom in dict.keys():
            dict[lfrom].add(lto)
        else:
            dict[lfrom] = set()
            dict[lfrom].add(lto)
        if not(lto in dict.keys()):
            dict[lto] = set()
    sites = list(dict.keys())
    A = np.array([sum([bv(l_to, sites) for l_to in dict[l_from]],
            np.zeros(len(sites)))
        for l_from in sites])
    return (sites, A)
```

Next, use the function from problem 2 to form the adjacency matrix, form the top-ten list using the 1-eigenvector, and compare to the power iteration method.

Comparison of methods

```
(ll,A) = adj_from_json("data.json")
n=len(ll)
C=make_transition(A,p=0.8)

print('Top␣list␣using␣eigenvector:')

l, v= np.linalg.eig(C)
w=np.real(v[:,0])
w/=w.sum()
maxnodes=[ll[i] for i in w.argsort()[-10:].astype(int)]
print(maxnodes)

print('Top␣list␣using␣power␣iterations:')

powers=np.arange(5)

for m in powers:
    w=np.linalg.matrix_power(C,m)@(np.ones(n)/n)

    maxnodes=[ll[i] for i in w.argsort()[-10:].astype(int)]
    print('␣m='+str(m)+':',
          maxnodes)

print('———————————————')
print('Varying p:')
for p in 0.1*(np.arange(9)+1):
    print(' p=',p)
    C=make_transition(A,p=p)
    l, v= np.linalg.eig(C)
    w=np.real(v[:,0])
    w/=w.sum()
    maxnodes=[ll[i] for i in w.argsort()[-10:].astype(int)]
    print('',maxnodes)
```

The top-ten list using the eigenvector versus power methods for $p = 0.8$ are given below:

```
Top list using eigenvector:
['Albatross', 'Carp', 'Gerbil', 'Fowl', 'Grouse', 'Rook', 'Squirrel', 'Donkey', 'Ant', 'Blue Whale']
Top list using power iterations:
 m=0: ['Bonobo','Rhinoceros','Amphibian','Beetle','Penguin','Basilisk','Heron','Takin','Fowl','Angelfish']
 m=1: ['Fowl', 'Gerbil', 'Rook', 'Salmon', 'Lemur', 'Grouse', 'Squirrel', 'Donkey', 'Ant', 'Blue Whale']
 m=2: ['Albatross', 'Carp', 'Gerbil', 'Fowl', 'Grouse', 'Rook', 'Squirrel', 'Donkey', 'Ant', 'Blue Whale']
 m=3: ['Albatross', 'Carp', 'Gerbil', 'Fowl', 'Grouse', 'Rook', 'Squirrel', 'Donkey', 'Ant', 'Blue Whale']
 m=4: ['Albatross', 'Carp', 'Gerbil', 'Fowl', 'Grouse', 'Rook', 'Squirrel', 'Donkey', 'Ant', 'Blue Whale'].
```

It only takes 2 iterations for the first column of C^m to give the correct ranking.
Next, we look at how changing p affects the ranking. Vary p from 0.1 to 0.9 in increments of 0.1.

```
Varying p:
 p= 0.1
 ['Bat', 'Fowl', 'Grouse', 'Carp', 'Gerbil', 'Albatross', 'Rook', 'Donkey', 'Ant', 'Blue Whale']
 p= 0.2
 ['Bat', 'Grouse', 'Fowl', 'Carp', 'Gerbil', 'Albatross', 'Rook', 'Donkey', 'Ant', 'Blue Whale']
 p= 0.30000000000000004
 ['Squirrel', 'Grouse', 'Fowl', 'Carp', 'Gerbil', 'Albatross', 'Rook', 'Donkey', 'Ant', 'Blue Whale']
 p= 0.4
 ['Squirrel', 'Grouse', 'Fowl', 'Carp', 'Albatross', 'Gerbil', 'Rook', 'Donkey', 'Ant', 'Blue Whale']
 p= 0.5
 ['Squirrel', 'Grouse', 'Carp', 'Albatross', 'Fowl', 'Gerbil', 'Rook', 'Donkey', 'Ant', 'Blue Whale']
 p= 0.6000000000000001
 ['Albatross', 'Carp', 'Grouse', 'Gerbil', 'Fowl', 'Squirrel', 'Rook', 'Donkey', 'Ant', 'Blue Whale']
 p= 0.7000000000000001
 ['Albatross', 'Carp', 'Gerbil', 'Fowl', 'Grouse', 'Squirrel', 'Rook', 'Donkey', 'Ant', 'Blue Whale']
 p= 0.8
 ['Albatross', 'Carp', 'Gerbil', 'Fowl', 'Grouse', 'Rook', 'Squirrel', 'Donkey', 'Ant', 'Blue Whale']
 p= 0.9
 ['Salmon', 'Gerbil', 'Lemur', 'Fowl', 'Rook', 'Grouse', 'Squirrel', 'Donkey', 'Ant', 'Blue Whale']
```

From these results, we can see a lot of variation in the top ranking. With $p = 0.3, 0.4, 0.5$, the results are fairly consistent. Similarly, they are fairly consistent for $p = 0.6, 0.7, 0.8$, though quite different from $0.3 \leq p \leq 0.5$. It's surprising that when $p = 0.9$, the top hit is Salmon, which is not featured in any of the other top rankings for any other value of p. The same is true of Bat, which is featured at the top ranking for $p = 0.1$ and $p = 0.2$, but is not in any of the other rankings. We also notice that Donkey, Ant, and Blue Whale are always the 8-10 rankings, regardless of p value.

7. The damping probability $0 < p \leq 1$ forces the `PageRank` transition matrix to satisfy the conclusion of the Perron-Frobenius Theorem. As we discussed above, from the point-of-view

of ranking web pages this quantity represents the probability that a random web-surfer gets bored and makes a random new choice of sites. In particular, this provides a reasonable self-contained explanation for the use of this `PageRank` transition matrix. On the other hand, one might hope to use the `PageRank` approach to rank other "linked material". For example, one might try to rank published academic papers by citation. More precisely: given a collection of academic papers, consider a directed graph whose nodes are the papers and for which there is a directed edge from paper A to paper B if paper A contains a citation to paper B. Suppose that each paper in the collection has at least one citation to another paper in the collection; under this assumption, the "naive" transition matrix T is stochastic. But this transition matrix may fail to satisfy the conclusion of the Perron-Frobenius Theorem. One can "fix" this problem as before by introducing a "damping probability" $0 < p \leq 1$ – i.e. by replacing T with the matrix

$$(1-p)T + \frac{p}{N}\mathbf{1}^{N \times N}$$

where N is the number of papers in the collection and $\mathbf{1}^{N \times N}$ denotes the $N \times N$ matrix all of whose entries are equal to 1. Do you think this is a reasonable proposal? What would be the explanation for the damping probability from the point-of-view of the model?

This is a reasonable proposal. Assume that we are researching a topic, and there's a finite set of papers that we've found on the topic, and the transition matrix describes the process of researching. With the above model, with probability $1-p$, the next paper we would look in would be cited in the paper we are currently looking at. With probability p/N, we'd just pick a paper at random from the pile. So we could think of $1-p$ as a probability of finding useful things in the paper we read (or one cited in it), and p/N as the probability that we don't find what we need, and randomly search.