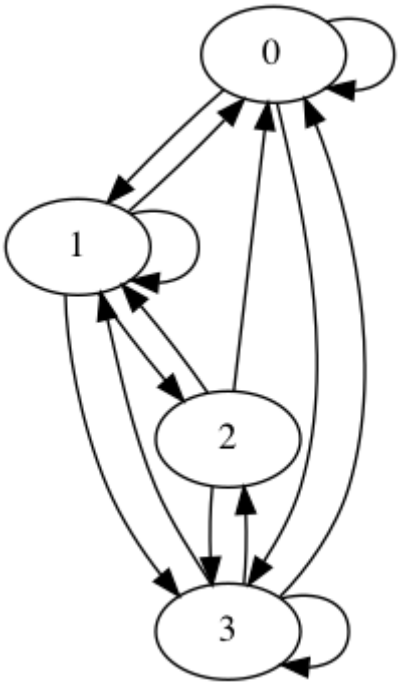
In [1]:
```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
%matplotlib inline

import sys
sys.path.append('./additional_files')
from graph import *
```

Review of graphs and adjacency matrices

How can we write the graph below as an adjacency matrix?

In [2]:
```
make_graph_adj_random(4)
```



Remember that:

$A_{ij} = 1$ if the node j has an outgoing edge (arrow) going to node i

$A_{ij} = 0$ otherwise

Hence the rows indicate incoming edges to a specific node and columns indicate outgoing edges from a specific node.

Write the adjacency matrix A of the above graph:

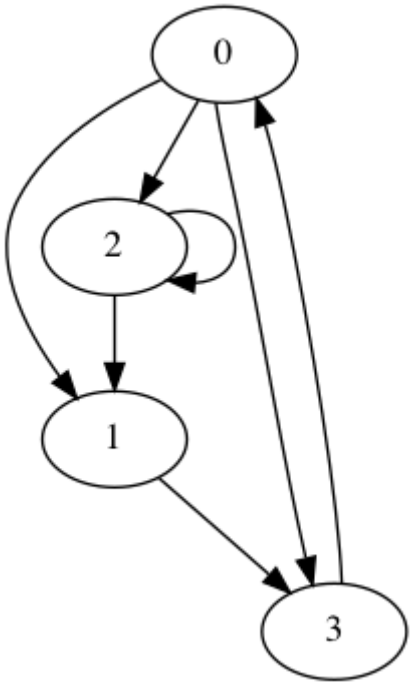
In [3]:
```
#clear
A = np.array([[0,1,1,1],[0,0,0,1],[0,1,1,0],[1,0,0,0]]).T
```

You can call the helper function:

```
graph.graph_matrix(mat, mat_label=None, show_weights=True, round_digits=3)
# mat: 2d numpy array of shape (n,n) with the adjacency matrix
# mat_label: 1d numpy array of shape (n,) with optional labels for the nodes
# show_weights: boolean - option to display the weights of the edges
# round_digits: integer - number of digits to display when showing the edge
 weights
```

to check if you get the same graph. Since in this example all the edge weights are zero, you should use `show_weights=False`

In [4]:
```
#clear
graph_matrix(A, show_weights=False)
```



Transition matrices

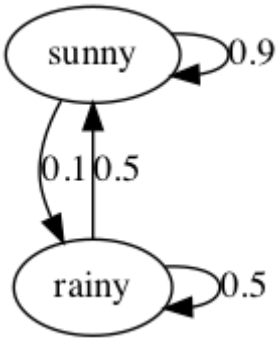
Using graphs to represent the transition from one state to the other

After collecting data about the weather for many years, you observed that the chance of a rainy day occurring after one rainy day is 50% and that the chance of a rainy day after one sunny day is 10%.

The graph that represents the transition from the weather on day 1 to the weather on day 2 can be expressed as an adjacency matrix, where the edge weights are the probabilities of weather conditions. We call that the transition matrix.

Write the transition matrix $\mathbf{A}$ for the weather observation described above, and use the helper function to plot the graph. Your graph will look better if you assign the labels for your nodes, for example, use the label `['sunny','rainy']`

In [5]:
```
#clear
A  = np.array([
    [  0.9, 0.5],
    [0.1,   0.5]
])
graph_matrix(A,['sunny', 'rainy'])
```



Properties of a transition (Markov) matrix

- A_{ij} entry of a transition matrix has the probability of transitioning from state j to state i
- Since the entries are probabilities, they are always non-negative real numbers, and the columns should add up to one.

The weather today is sunny. What is the probability of sunny day tomorrow?

The answer to this question is quite trivial, and in this example, we can directly get that information from the graph. But how could we obtain the same as a matrix-vector multiplication?

Write the numpy array $\mathbf{x0}$ representing the current state, which is the weather today. Your vector should be consistent with the transition matrix. If the first column of $\mathbf{A}$ corresponded to transitioning **from** a sunny day, the first entry of the state vector should be sunny.

In [6]:
```
#clear
x0 = np.array([1,0])
```

You can now obtain the probabilities for tomorrow weather by performing a matrix-vector multiplication:

In [7]:
```
#clear
x1 = A@x0
print(x1)
```

```
[0.9 0.1]
```

The weather today (Thursday) is sunny. What is the probability of rain on Sunday? Should I keep my plans for a barbecue?

In [8]:
```
#clear
x = np.array([0.5,0.5])
x = A@x # Friday
x = A@x # Saturday
x = A@x # Sunday
print('Probability of rain is ', x[1])
```

```
Probability of rain is  0.188
```

What is the probability of sunny days in the long run?

Let's run power iteration for the equivalent of 20 days. We will store all the state vectors $\mathbf{x}$ as columns of a matrix, so that we can plot the results later.

We will store all iterations in `allx` in order to view the running probabilities later.

In [9]:
```
its = 40
allx = np.zeros((2,its))
```

We will start with the initial state of rainy day:

In [10]:
```
x = np.array([0.5,0.5])
allx[:,0] = x
```

What are the probabilities after 20 days?

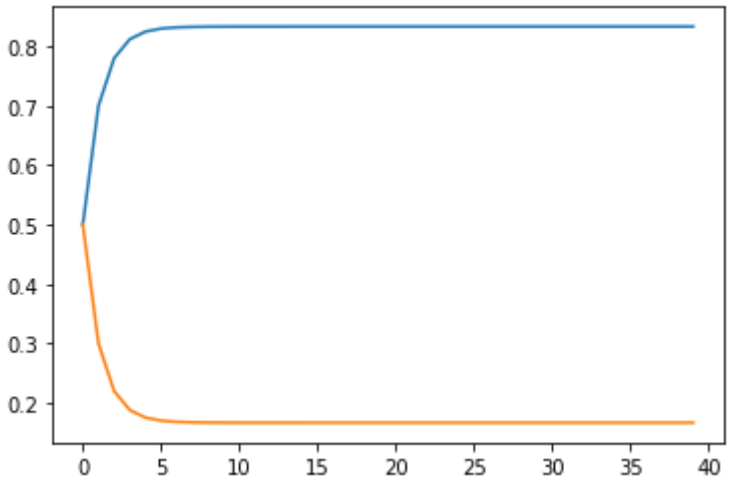
In [11]:
```
#clear
for k in range(1,its):
    x = A@x
    allx[:,k] = x
x
```

Out[11]:
```
array([0.83333333, 0.16666667])
```

In [12]:
```
print('Probabilities of initial state:', allx[:,0])
print('Probabilities after 20 days:', allx[:,-1])
plt.plot(allx.T)
plt.xlabel('')
```

```
Probabilities of initial state: [0.5 0.5]
Probabilities after 20 days: [0.83333333 0.16666667]
```

Out[12]:
```
Text(0.5, 0, '')
```



What if we were to start with a random current state? Would the weather probabilities change in the long run? Write a similar code snippet, but now starting at a random initial state. Remember that the sum of the probabilities has to be equal to 1 (or that columns and state vectors should sum to 1). Think about normalizing. Which norm would satisfy this property?

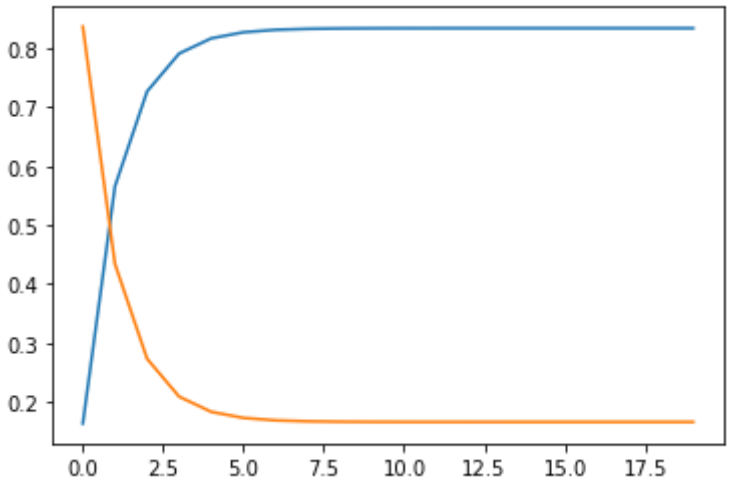
In [13]:
```
#clear
x = np.random.rand(2)
x = x/la.norm(x,1)
print(x)

its = 20
allx = np.zeros((2,its))
allx[:,0] = x

for k in range(1,its):
    x = A@x
    allx[:,k] = x

print(x)
```

```
[0.1639922 0.8360078]
[0.83333331 0.16666669]
```

In [14]:
```
print('Probabilities of initial state:', allx[:,0])
print('Probabilities after 20 days:', allx[:,-1])
plt.plot(allx.T)
plt.xlabel('')
```

```
Probabilities of initial state: [0.1639922 0.8360078]
Probabilities after 20 days: [0.83333331 0.16666669]
```

Out[14]:
```
Text(0.5, 0, '')
```



You can run your code snippet above a few times. What do you notice?

Summarizing:

- You started with an initial weather state $\mathbf{x_0}$.
- Using the transition matrix, you can get the weather probability for the next day: $\mathbf{A x_0} = \mathbf{x_1}$.
- In general, you can write $\mathbf{A x_n} = \mathbf{x_{n+1}}$.
- Predictions for the weather on more distant days are increasingly inaccurate.
- Power iteration will converge to a **steady-state** vector, that gives the weather probabilities in the long-run.

$$\mathbf{A x^*} = \mathbf{x^*}$$

- $\mathbf{x^*}$ is the long-run equilibrium state, or the eigenvector corresponding to the eigenvalue $\lambda = 1$.
- This steady-state condition $\mathbf{x^*}$ does not depend on the initial state.

Example 2: What are students doing during CS 357 lectures?

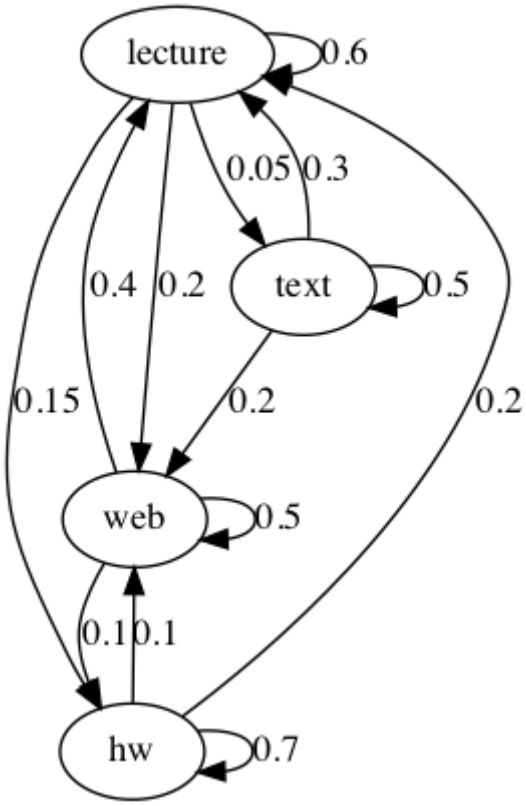
Consider the following graph of states. Assume this is a model of the behavior of a student after 10 minutes of a lecture. For example, if a student at the beginning of a lecture is participating in the class activity, there is a probability of 20% that they will be surfing the web after 10 minutes.

Write the transition matrix $\mathbf{A}$ corresponding to the graph above. Build your matrix so that the columns are given in the following order:

```
activity_names = ['lecture', 'web', 'hw', 'text']
```

Plot the graph using the `graph_matrix` helper function, to make sure you are indeed building the correct matrix. You can use the list above to label the nodes.

In [15]:
```
#clear

activity_names = ['lecture', 'web', 'hw', 'text']

A  = np.array([
    [0.6,  0.4, 0.2, 0.3],
    [0.2, 0.5, 0.1, 0.2],
    [0.15, 0.1,  0.7, 0.0],
    [0.05, 0, 0  , 0.5]
])

graph_matrix(A,activity_names)
```



Build the initial state vector $\mathbf{x}$. Recall that this array should follow the same order as the matrix, here defined in `activity_names`. In the initial state (suppose that we are at the 0 minute mark of the lecture, i.e. the beginning of the lecture), students have an 80% probability of participating in lecture, 10% probability of texting friends and 10% probability of surfing the web.

In [16]:
```
#clear
x0 = np.array([0.8,0.1,0,0.1])
```

What is the probability that students will be working on their HW after 10 minutes?

In [17]:
```
#clear
x = x0
x = A@x
print(x)
print("Probability of working on HW after 10 minutes is:", x[2])
```

```
[0.55 0.23 0.13 0.09]
Probability of working on HW after 10 minutes is: 0.13
```

What is the probability that students will be sending text messages after 30 minutes?

In [18]:
```
#clear
x = np.random.random(4)
x = x/la.norm(x,1)

#x0 = np.array([0.8,0.1,0,0.1])
for k in range(20):
    x=A@x
    print("x at",(k+1)*10,"min = ",x)
print("Probability of surfing the web after 30 minutes is:", x[3])
```

```
x at 10 min =  [0.42442737 0.30231178 0.24670192 0.02655894]
x at 20 min =  [0.4328892  0.26602334 0.26658662 0.03450084]
x at 30 min =  [0.42981043 0.25314834 0.27814635 0.03889488]
x at 40 min =  [0.42644333 0.24812987 0.28448884 0.04093796]
x at 50 min =  [0.4242971  0.24599008 0.28792168 0.04179115]
x at 60 min =  [0.42309597 0.24500486 0.28978875 0.04211043]
x at 70 min =  [0.4224504  0.24452258 0.290817   0.04221001]
x at 80 min =  [0.42210568 0.24427507 0.29139172 0.04222753]
x at 90 min =  [0.42192004 0.24414335 0.29171756 0.04221905]
x at 100 min =  [0.42181859 0.24407125 0.29190464 0.04220553]
x at 110 min =  [0.42176224 0.24403091 0.29201316 0.04219369]
x at 120 min =  [0.42173045 0.24400796 0.29207664 0.04218496]
x at 130 min =  [0.42171227 0.24399472 0.29211401 0.042179  ]
x at 140 min =  [0.42170175 0.24398702 0.29213612 0.04217511]
x at 150 min =  [0.42169562 0.24398249 0.29214925 0.04217264]
x at 160 min =  [0.42169201 0.24397982 0.29215706 0.0421711 ]
x at 170 min =  [0.42168988 0.24397824 0.29216173 0.04217015]
x at 180 min =  [0.42168861 0.2439773  0.29216452 0.04216957]
x at 190 min =  [0.42168786 0.24397674 0.29216618 0.04216922]
x at 200 min =  [0.42168741 0.2439764  0.29216718 0.042169  ]
Probability of surfing the web after 30 minutes is: 0.04216900098315762
```

Would your answer above change if you were to start from a different initial state? Try for example x0 = [0, 0.2, 0.1, 0.7].

In [19]:
```
#clear
x = np.array([0, 0.2, 0.1, 0.7])
for k in range(3):
    x=A@x
    print("x at",(k+1)*10,"min = ",x)
print("Probability of surfing the web after 30 minutes is:", x[3])
```

```
x at 10 min =  [0.31 0.25 0.09 0.35]
x at 20 min =  [0.409  0.266  0.1345 0.1905]
x at 30 min =  [0.43585 0.26635 0.1821  0.1157 ]
Probability of surfing the web after 30 minutes is: 0.1157
```

Steady-state will not depend on the initial state, but before achieving steady-state, the probabilities will be different!

Get the steady-state for this problem:

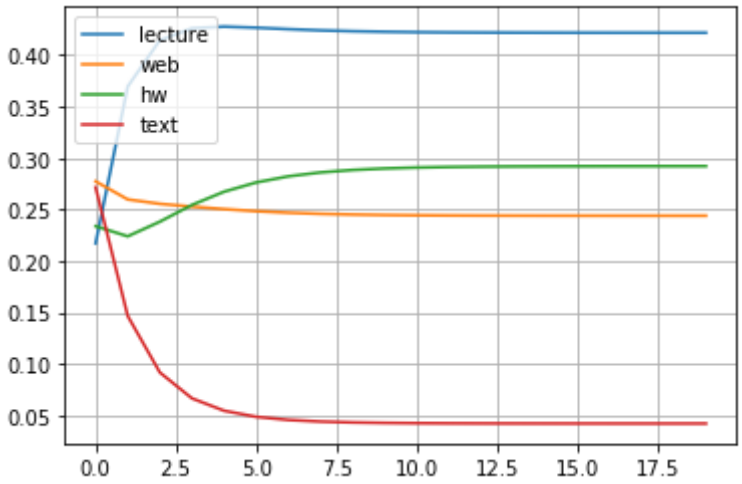
- Start from a random initial state.
- Make sure the probability of the initial state sums to 1.
- Assume 20 iterations of power method

In [20]:
```
its = 20
allx = np.zeros((4,its))
```

In [21]:
```
#clear
x0 = np.random.rand(4)
x0 = x0/la.norm(x0,1)

x = np.copy(x0)
allx[:,0] = x

for k in range(1,its):
    x = A.dot(x)
    allx[:,k] = x

print(x)
```

```
[0.4216931  0.24398067 0.29215435 0.04217188]
```

In [22]:
```
plt.plot(allx.T)
plt.legend(activity_names)
plt.grid()
```



Do you think you achieved steady-state at iteration 40?

Note that iteration 40 corresponds to the 400th minute of a lecture, which in this case does not exist (luckily? :-))

Does the class achieve steady-state at the end of lecture (about 70 minutes or 7 steps)?

Use `numpy.linalg.eig` to see how the steady-state (solution of power iteration) is the eigenvector of $\mathbf{A}$ corresponding the the eigenvalue 1. The eigenvectors are given in the columns of $\mathbf{U}$.

In [23]:
```
lambdas, U = la.eig(A)
print(lambdas)
print(U[:,0])
```

```
[1.         0.26339746 0.6        0.43660254]
[0.7402746  0.42830173 0.51290455 0.07402746]
```

Not what you expected? Remember that if $\mathbf{u}$ is an eigenvector of $\mathbf{A}$, then $\alpha\mathbf{u}$ is also en eigenvector. Can you show that the eigenvector from the power iteration solution is the same as the one obtained using `numpy.linalg.eig` ?

In [24]:
```
#clear
uvec = U[:,0]
uvec = uvec/la.norm(uvec,1)
print(uvec)
```

```
[0.42168675 0.2439759  0.29216867 0.04216867]
```

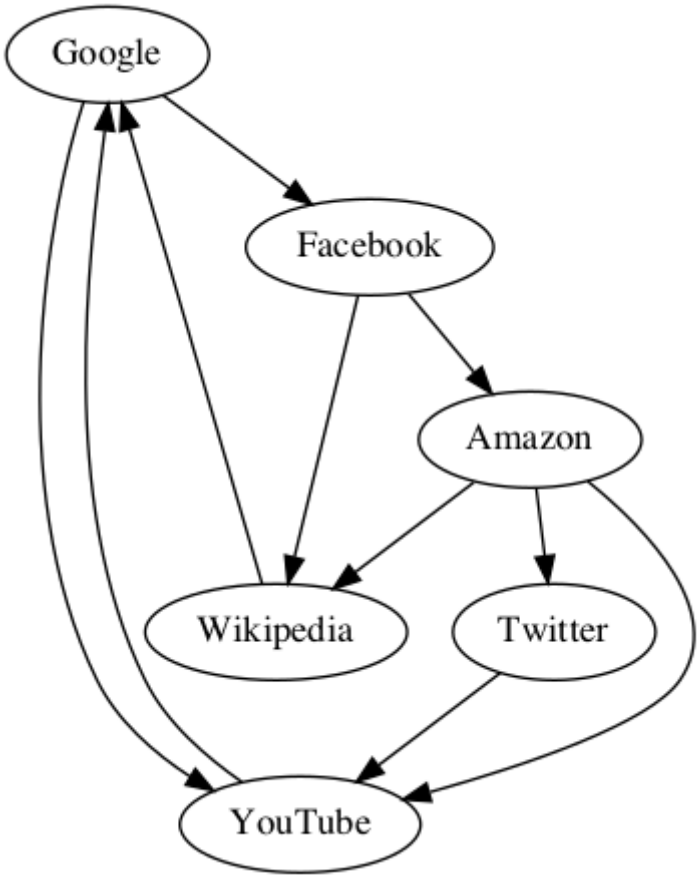
Example 3: Page Rank

Suppose we have 6 different websites that we want to rank. They appear in the dictionary `name_mapping`, that indicates the website string name for each node in the graph (or position in the transition matrix).

In [25]:
```
name_mapping={0:'Google', 1:'Facebook', 2:'Amazon', 3:'Wikipedia', 4:'Twitter', 5:'YouTube'}
```

The list `edges` contains the outgoing link information. Each list entry `[x,y]` indicates that a node x (or website x) has a link to node y (or website y). The code snippet below construct the adjacency matrix, and plots the corresponding graph.

In [26]:
```
edges = [
    [0,1], [0,5],
    [1,2], [1,3],
    [2,3], [2,4], [2,5],
    [3,0],
    [4,5],
    [5,0]]

A = np.zeros((6,6))
for i, j in edges:
    A[j, i] = 1

graph_matrix(A, list(name_mapping.values()), show_weights=False)
```



Definition of the PageRank model:

“PageRank can be thought of as a model of user behavior. We assume there is a random surfer who is given a web page at random and keeps clicking on links, never hitting “back”…”

http://infolab.stanford.edu/~backrub/google.html (http://infolab.stanford.edu/~backrub/google.html)

Let's put numbers to the statement above and see how it applies to the example above. Suppose a random surfer is initially at website 0. His initial state would be:

[1,0,0,0,0,0]

The web surfer will click on links on current websites, and he will keep doing this until achieving steady-state (he will never hit "back button"). So for the example above, if he is on the Google website, he will either be going to Facebook or YouTube next. Since there are two links, we will assume that each website has the same probability of being the next one, so there is a probability of 50% he will go to Facebook and 50% he will go to YouTube.

We now need to modify the adjacency matrix above in order to get a matrix that satisfy the properties of the Markov matrix, specifically the one that the column sum is equal to 1 (here representing the probabilities of the outgoing links).

Here is what your adjacency matrix looks like:

```
In [27]: A
```

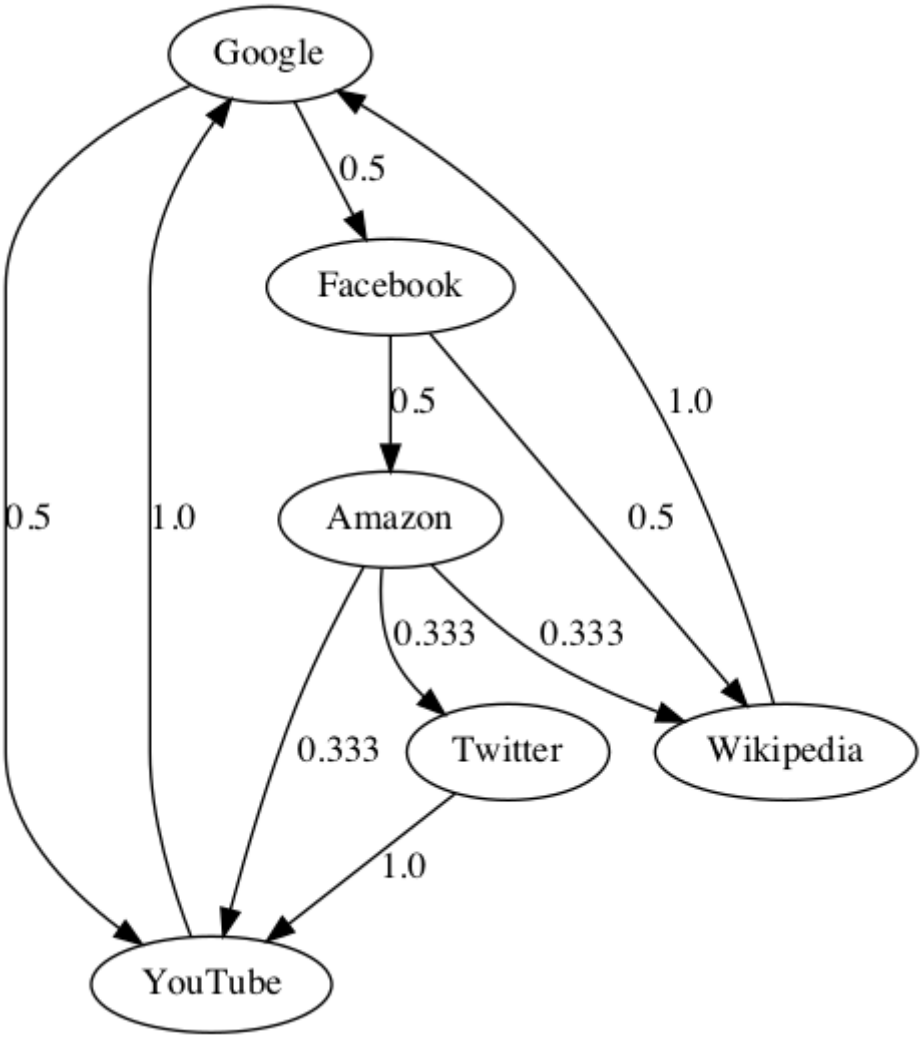
```
Out[27]: array([[0., 0., 0., 1., 0., 1.],
                [1., 0., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0., 0.],
                [0., 1., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0., 0.],
                [1., 0., 1., 0., 1., 0.]])
```

How should you change the matrix `A`, to model the behavior described above? Make the Markov matrix M.

In [28]:
```
#clear
n = A.shape[0]
M =  A / A.sum(axis=0)
M
```

Out[28]:
```
array([[0.        , 0.        , 0.        , 1.        , 0.        ,
        1.        ],
       [0.5       , 0.        , 0.        , 0.        , 0.        ,
        0.        ],
       [0.        , 0.5       , 0.        , 0.        , 0.        ,
        0.        ],
       [0.        , 0.5       , 0.33333333, 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.33333333, 0.        , 0.        ,
        0.        ],
       [0.5       , 0.        , 0.33333333, 0.        , 1.        ,
        0.        ]])
```

In [29]:
```
graph_matrix(M, list(name_mapping.values()))
```



Get the steady-state for this problem:

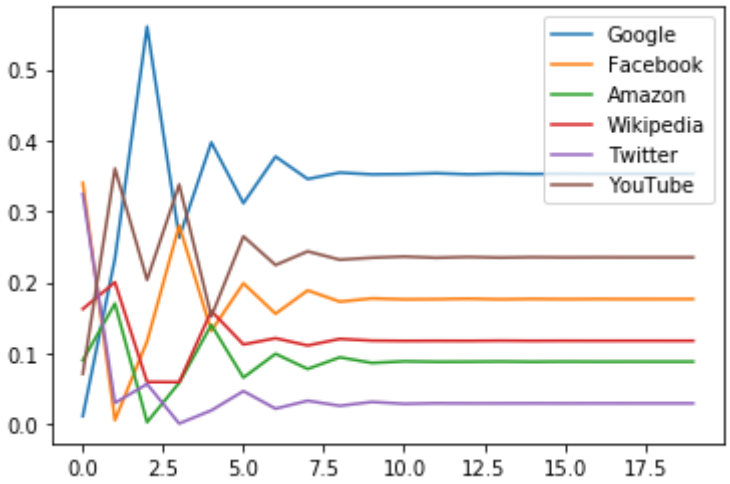
- Start from a random initial state.
- Make sure the probability of the initial state sums to 1.
- Assume 20 iterations of power method

In [30]:
```
#clear
x0 = np.random.rand(n)
x0N=x0/la.norm(x0,1)

x = x0N.copy()
its = 20
allx = np.zeros((n,its))
allx[:,0] = x

for k in range(1,its):
    x = M.dot(x)
    allx[:,k] = x

print("x = ",x)
```

```
x =  [0.35293884 0.17646901 0.08823985 0.11764654 0.02940669 0.2352990
7]
```

In [31]:
```
plt.plot(allx.T)
plt.legend(list(name_mapping.values()))
```

Out[31]:
```
<matplotlib.legend.Legend at 0x11fc2ab90>
```



At steady-state, what is the most likely page the user will end up at, when starting from a random page?

Creating the list of strings `ranking` with the rank of the websites (the string names), starting from the mostly likely the user will end up at, to the least likely.

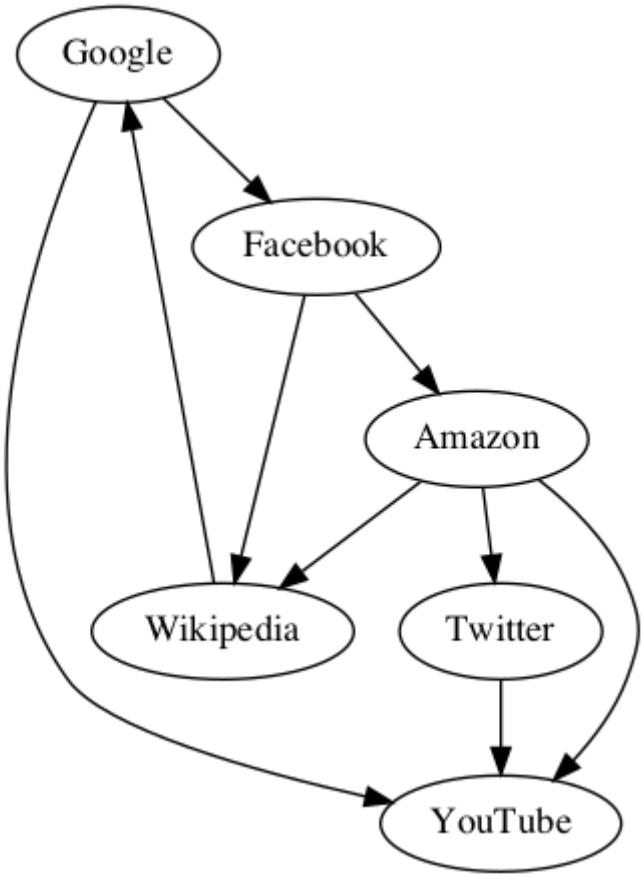
In [32]:
```
#clear
ranking = []
for index in np.argsort(x)[::-1]:
    ranking.append(name_mapping[index])

ranking
```

Out[32]:
```
['Google', 'YouTube', 'Facebook', 'Wikipedia', 'Amazon', 'Twitter']
```

What happens when the web surfer ends up in a website without any outgoing link?

Let's repeat the example above, with a simple change to the outgoing links. Page 5 (YouTube) no longer has an outgoing link.

In [33]:
```
edges = [
    [0,1], [0,5],
    [1,2], [1,3],
    [2,3], [2,4], [2,5],
    [3,0],
    [4,5]]

A = np.zeros((6,6))
for i, j in edges:
    A[j, i] = 1

graph_matrix(A, list(name_mapping.values()), show_weights=False)
```



Note that you can no longer use the method above to obtain the Markov matrix, since the sum of the fifth column is now zero.

In [34]:
```
A
```

Out[34]:
```
array([[0., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 1., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [1., 0., 1., 0., 1., 0.]])
```

Once the web surfer reaches a page without outgoing links, we can assume that he will not stay on that webpage forever!

We then assume that the web surfer will move to any of the other webpages with equal probability. Construct the Markov matrix using this assumption.

In [35]:
```
#clear
n = A.shape[0]
for i,s in enumerate(A.sum(axis=0)):
    if s != 0:
        M[:,i] = A[:,i] / s
    else:
        M[:,i] = 1/n

M
```

Out[35]:
```
array([[0.        , 0.        , 0.        , 1.        , 0.        ,
        0.16666667],
       [0.5       , 0.        , 0.        , 0.        , 0.        ,
        0.16666667],
       [0.        , 0.5       , 0.        , 0.        , 0.        ,
        0.16666667],
       [0.        , 0.5       , 0.33333333, 0.        , 0.        ,
        0.16666667],
       [0.        , 0.        , 0.33333333, 0.        , 0.        ,
        0.16666667],
       [0.5       , 0.        , 0.33333333, 0.        , 1.        ,
        0.16666667]])
```

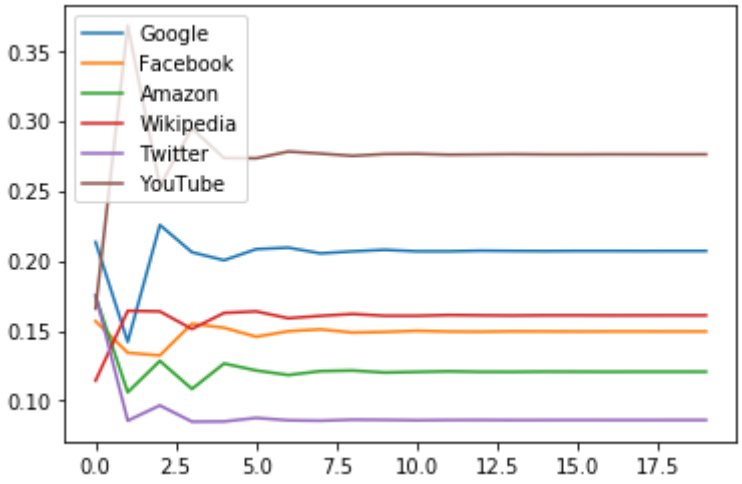
Get the steady-state for this problem:

- Start from a random initial state.
- Make sure the probability of the initial state sums to 1.
- Assume 20 iterations of power method

Did your ranking changed?

In [36]:
```
#clear
x0 = np.random.rand(n)
x0N=x0/la.norm(x0,1)

x = x0N.copy()
its = 20
allx = np.zeros((n,its))
allx[:,0] = x

for k in range(1,its):
    x = M.dot(x)
    allx[:,k] = x


plt.plot(allx.T)
plt.legend(list(name_mapping.values()))

#clear
ranking = []
for index in np.argsort(x)[::-1]:
    ranking.append(name_mapping[index])

ranking
```

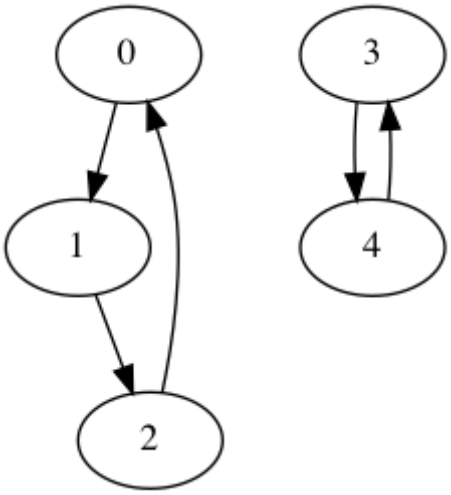
Out[36]:
```
['YouTube', 'Google', 'Wikipedia', 'Facebook', 'Amazon', 'Twitter']
```



One remaining issue!

The Markov matrix does not guarantee a unique solution! Let's take a look at this example:

In [37]:
```
B = np.array([[0,0,1,0,0],[1,0,0,0,0],[0,1,0,0,0],[0,0,0,0,1],[0,0,0,1,0
]])
graph_matrix(B,  show_weights=False)
```



The matrix `B` has two eigenvectors corresponding to the same eigenvalue of 1.

In [38]:
```
l,v = la.eig(B)
print('The eigenvalues are:')
print(l)
```

```
The eigenvalues are:
[-0.5+0.8660254j -0.5-0.8660254j  1. +0.j         1. +0.j
 -1. +0.j       ]
```

Some of the eigenvalues are complex, but we are interested in the two eigenvectors corresponding to the eigenvalue 1.

In [39]:
```
v[:,2]/la.norm(v[:,2],1)
```

Out[39]:
```
array([-0.33333333+0.j, -0.33333333+0.j, -0.33333333+0.j,  0.        +
0.j,
        0.        +0.j])
```

In [40]:
```
v[:,3]/la.norm(v[:,3],1)
```

Out[40]:
```
array([0. +0.j, 0. +0.j, 0. +0.j, 0.5+0.j, 0.5+0.j])
```

Hence both

$$\mathbf{x}^* = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0.5 \\ 0.5 \end{bmatrix} \quad \text{and} \quad \mathbf{x}^* = \begin{bmatrix} 0.33 \\ 0.33 \\ 0.33 \\ 0 \\ 0 \end{bmatrix}$$

are possible solutions!

Perron-Frobenius theorem (CIRCA 1910):

If $\mathbf{M}$ is a Markov matrix with ***all positive entries***, then $\mathbf{M}$ has unique steady-state vector $\mathbf{x}^*$.

Definition of the PageRank model (complete):

“PageRank can be thought of as a model of user behavior. We assume there is a random surfer who is given a web page at random and keeps clicking on links, never hitting 'back', but **eventually gets bored and starts on another random page**."

To model the behavior of a web surfer getting bored, the proposed **Google matrix** is defined as:

$$G = \alpha \mathbf{M} + (1 - \alpha) \frac{1}{n} \mathbf{1}$$

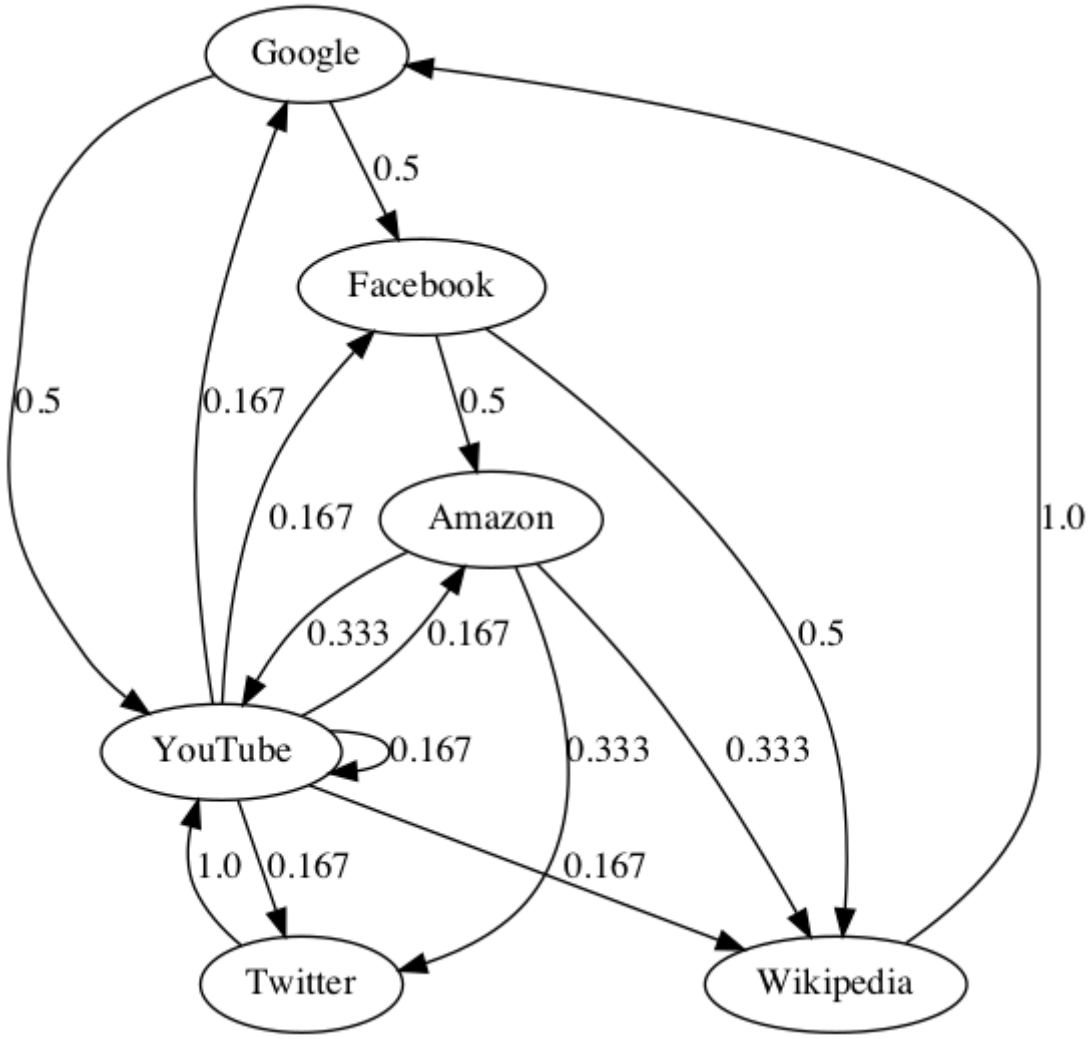
where $\mathbf{1}$ is a matrix of ones with same dimension n as the Markov matrix $\mathbf{M}$. We divide $\mathbf{1}$ by n to enforce that the columns sum to 1. In this model, a web surfer clicks on a link on the current page with probability α and opens a random page with probability $(1 - \alpha)$, where $0 < \alpha < 1$. A typical value for α is 0.85. Note that the Google matrix $\mathbf{G}$ has all entries greater than zero, and guarantees a unique solution.

Construct the Google matrix $\mathbf{G}$ for the Markov matrix above, using $\alpha = 0.85$:

In [41]:
```
print(M)
```

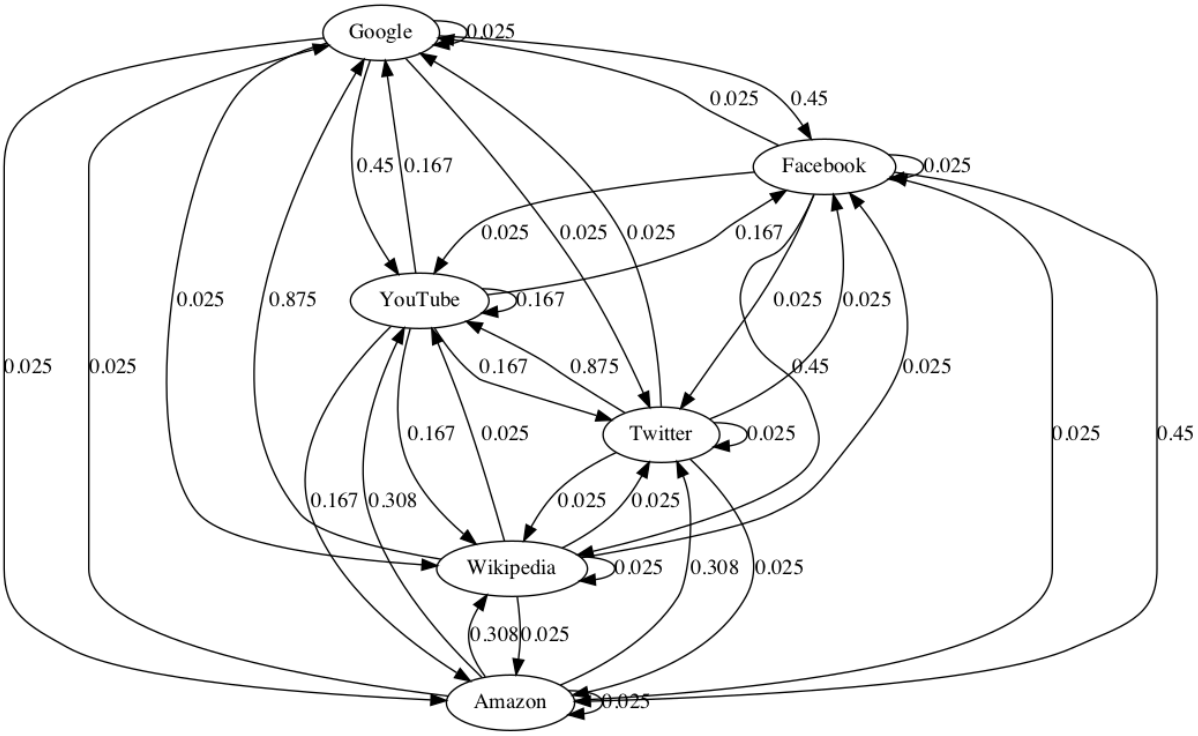
```
[[0.         0.         0.         1.         0.         0.16666667]
 [0.5        0.         0.         0.         0.         0.16666667]
 [0.         0.5        0.         0.         0.         0.16666667]
 [0.         0.5        0.33333333 0.         0.         0.16666667]
 [0.         0.         0.33333333 0.         0.         0.16666667]
 [0.5        0.         0.33333333 0.         1.         0.16666667]]
```

In [42]:
```
graph_matrix(M, list(name_mapping.values()))
```



In [43]:
```
#clear
alpha = 0.85
G = 0.85*M + 0.15*np.ones((n,n))/n
```

In [44]:
```
graph_matrix(G, list(name_mapping.values()))
```

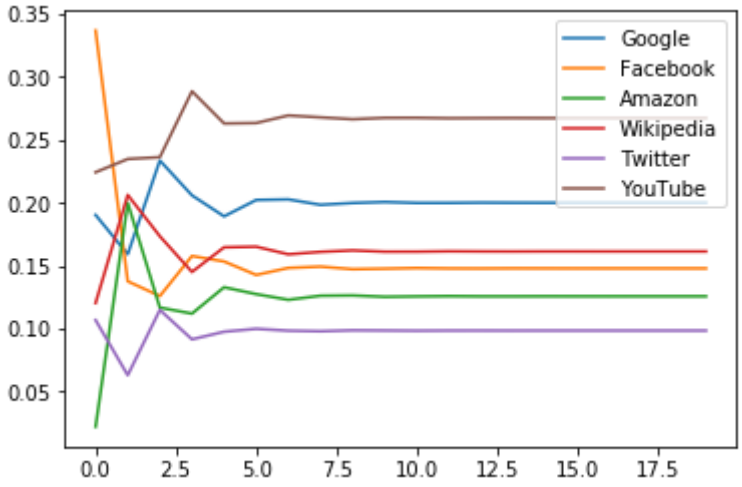


Get the steady-state for this problem:

- Start from a random initial state.
- Make sure the probability of the initial state sums to 1.
- Assume 20 iterations of power method

In [45]:
```
#clear
x0 = np.random.rand(n)
x0N=x0/la.norm(x0,1)

x = x0N.copy()
its = 20
allx = np.zeros((n,its))
allx[:,0] = x

for k in range(1,its):
    x = G.dot(x)
    allx[:,k] = x


plt.plot(allx.T)
plt.legend(list(name_mapping.values()))

#clear
ranking = []
for index in np.argsort(x)[::-1]:
    ranking.append(name_mapping[index])

ranking
```

Out[45]:
```
['YouTube', 'Google', 'Wikipedia', 'Facebook', 'Amazon', 'Twitter']
```



Overall complexity of PageRank problem

In general, for the dense Google matrix with shape (n, n), the cost of each matrix-vector multiplication during the power iteration method would be $O(n^2)$.

$$\mathbf{x}_{i+1} = \mathbf{G}\,\mathbf{x}_i$$

In a PageRank problem, n is very large! But let's look at each part of the Google matrix:

$$\mathbf{G} = \alpha\,\mathbf{M} + (1-\alpha)\,\frac{1}{n}\mathbf{1}$$

The Markov Matrix for the PageRank problem will likely be very sparse, since webpages will only have a few number of outgoing links. Hence, the calculation

$$\mathbf{x}_{i+1} = \mathbf{M}\,\mathbf{x}_i$$

will take advantage of sparse operations, which can be in general approximated with $O(n)$ complexity.

We need to look at the complexity of performing

$$\mathbf{x}_{i+1} = \mathbf{1}\,\mathbf{x}_i$$

Since $\mathbf{1}$ is a dense matrix, some would quickly jump to the conclusion that the complexity is $O(n^2)$. But it is not! Can you think of how this operation can be done efficiently?