

23-Networks-II-Centrality

December 5, 2017

1 Centrality

An important question in the analysis of networks is to understand the relative "importance" of the nodes in the network.

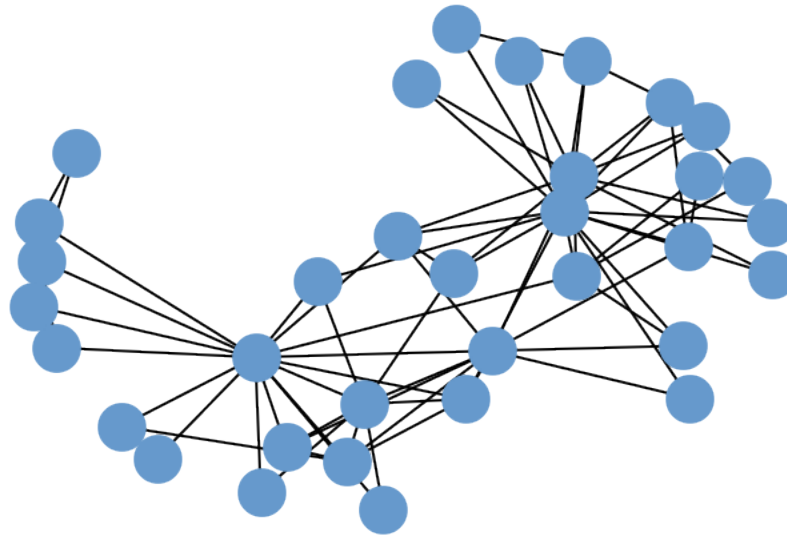
For example:

- in a social network, who are the most influential individuals?
- in the Web, which pages are more informative?
- in road network, which intersections are most heavily used?

The key idea is that the **structure** of the network should give us some information about the relative importance of the nodes in the network.

Recall Zachary's Karate Club:

```
In [104]: Gk=nx.karate_club_graph()
          np.random.seed(2)
          pos = nx.spring_layout(Gk)
          nx.draw_networkx(Gk, pos=pos, with_labels=False, node_color='#6699cc')
          plt.axis('off');
```



Do some nodes in the network have a special role?

Are some nodes more "important" than others?

These are often termed questions of **centrality** (or **prestige**).

Today we'll study metrics for these notions. We'll work with a graph G .

We will study three basic notions of centrality:

1. "Closeness" Centrality: A central node is close to all others.
2. "Betweenness" Centrality: A central node is on many paths through the network.
3. "Status" Centrality: A central node is connected to other central nodes.

1.1 Closeness Centrality

The closeness centrality of a node i is an indicator of the proximity between i and all the other nodes in the graph.

The standard way of formulating this is:

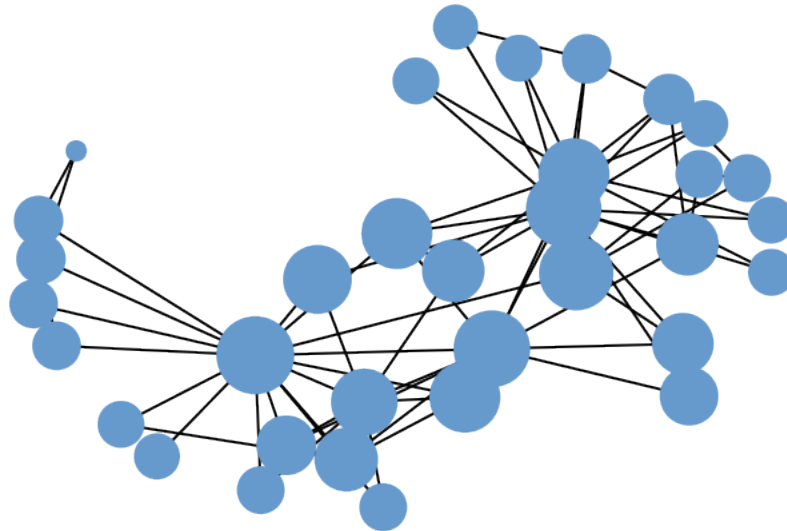
$$\text{closeness}(i) = \frac{1}{\sum_{j \in V} d(i, j)}.$$

$d(i, j)$ is the shortest path distance between i and j in G . (Although other dissimilarity measures could be used.)

It is assumed that G is connected.

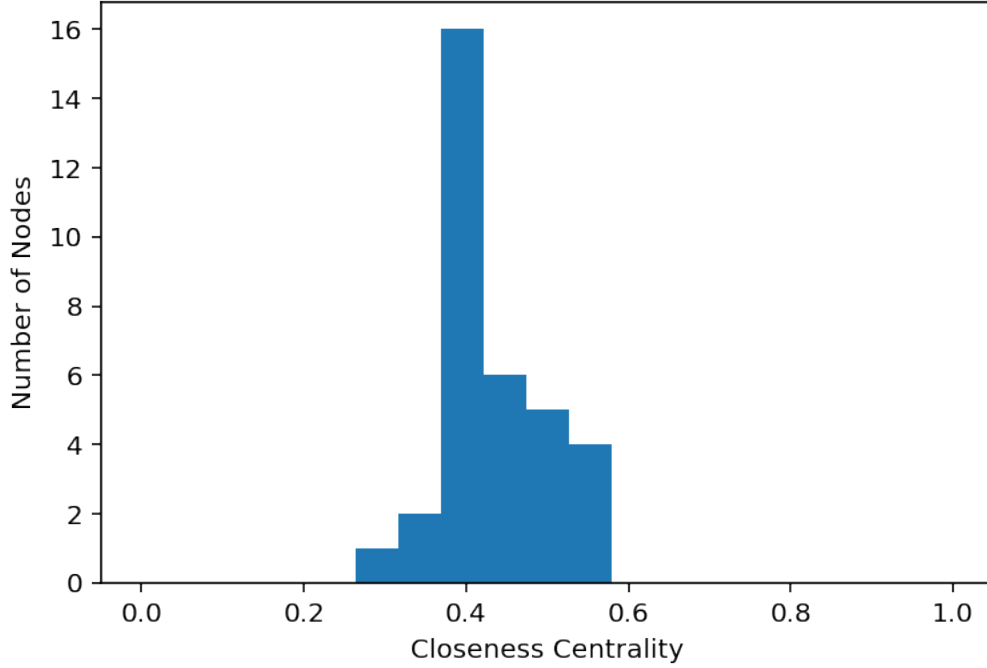
```
In [100]: # plotting the graph
cent = np.array(list(nx.closeness_centrality(Gk).values()))
scaler = MinMaxScaler((50,800))
scaledC = scaler.fit_transform(cent[:,np.newaxis])
```

```
nx.draw_networkx(Gk, node_size=scaledC, pos=pos, with_labels=False, node_color='#6699cc',  
plt.axis('off');
```



In this graph, most nodes are close to most other nodes. However we can see that some nodes are slightly more central than others.

```
In [75]: plt.hist(cent,bins=np.linspace(0,1,20))  
plt.xlabel('Closeness Centrality')  
_=plt.ylabel('Number of Nodes')
```



1.2 Betweenness Centrality

The next notion of centrality correlates to "on many paths."

If we picture the network as a conduit for information of some sort, then betweenness captures how important a node is to the communication process (or "how much" information passes through the node).

First, let's consider the case in which there is only one shortest path between any pair of nodes. Then, the betweenness centrality of node i is the number of shortest paths that pass through i .

$$\text{betweenness}(i) = \sum_{i \neq j \neq k \in V} \begin{cases} 1 & \text{if path from } j \text{ to } k \text{ goes through } i \\ 0 & \text{otherwise} \end{cases}$$

We can convert this to a value between 0 and 1 by dividing by $N(N-1)/2$.

In a general graph, there may be multiple shortest paths between j and k .

To handle this, we work in terms of the *dependency* of i on the paths between j and k :

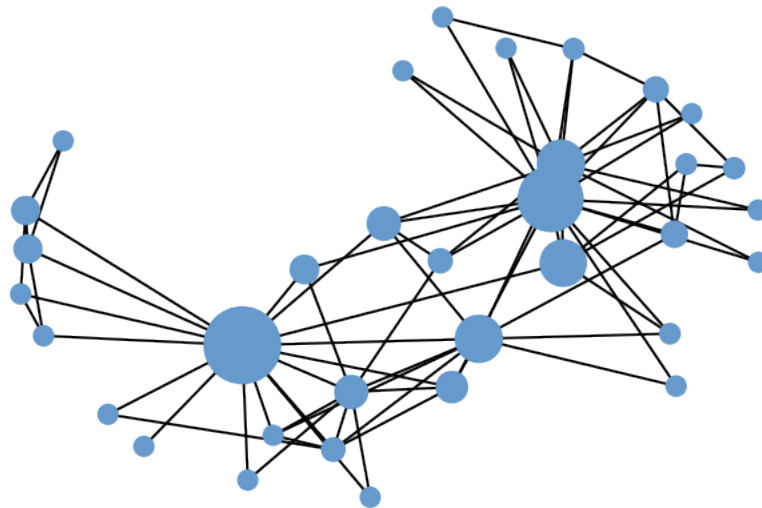
$$\text{dependency}(j, k | i) = \frac{\sigma(j, k | i)}{\sigma(j, k)}$$

where $\sigma(j, k | i)$ is the number of shortest paths between j and k that pass through i , and $\sigma(j, k)$ is the total number of shortest paths between j and k .

Then:

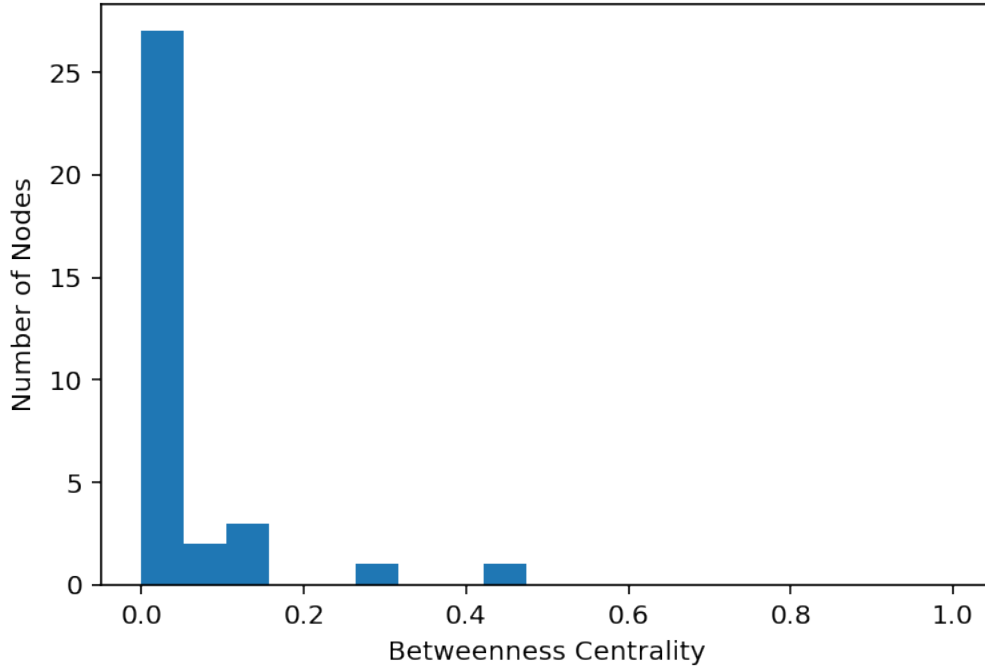
$$\text{betweenness}(i) = \sum_{i \neq j \neq k \in V} \text{dependency}(j, k | i)$$

```
In [76]: # plotting the graph
cent = np.array(list(nx.betweenness centrality(Gk).values()))
scaler = MinMaxScaler((50,800))
scaledC = scaler.fit_transform(cent[:,np.newaxis])
nx.draw_networkx(Gk, node_size=scaledC, pos=pos, with_labels=False, node_color='#6699cc')
plt.axis('off');
```



Here we can see more clearly the "importance" of two or three key members of the karate club.

```
In [77]: plt.hist(cent,bins=np.linspace(0,1,20))
plt.xlabel('Betweenness Centrality')
_=plt.ylabel('Number of Nodes')
```



1.3 Status Centrality

The third notion of centrality is more subtle.

In this context, one often talks of "prestige" rather than "centrality." But the concepts are related.

The idea of **status** centrality is that "high status" nodes are those that are connected to "high status" nodes.

If you think this definition is circular, you are right!

Nonetheless, it leads to some metrics that are quite well defined and not hard to compute.

Eigenvector Centrality.

Our first form of status centrality derives directly from the intuitive notion that high status nodes are those that associate with other high status nodes.

So in this case we say that the centrality of a node is just the sum of the centrality of its neighbors, times some factor.

That is,

$$Ev(i) = 1/\lambda \sum_{(i,j) \in E} Ev(j).$$

This defines a recursive formulation for $Ev(i)$ which may or may not converge, depending on λ .

In fact, we can rewrite this in terms of the adjacency matrix A where $a_{ij} = 1$ if $(i, j) \in E$.

$$\lambda Ev(i) = \sum_j a_{ij} Ev(j).$$

If we think of Ev as a column vector (say, \mathbf{e}), then this is:

$$A\mathbf{e} = \lambda\mathbf{e}$$

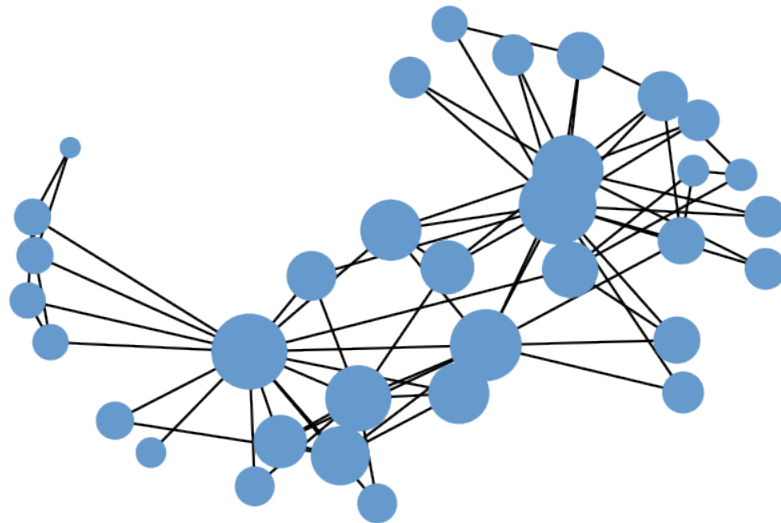
which tells us that \mathbf{e} must be an eigenvector of A .

There is a good argument for choosing λ as the largest eigenvalue of A . In this case all the entries in \mathbf{e} will have the same sign.

By convention we use the absolute value of the entries of \mathbf{e} as each node's **eigenvector centrality**.

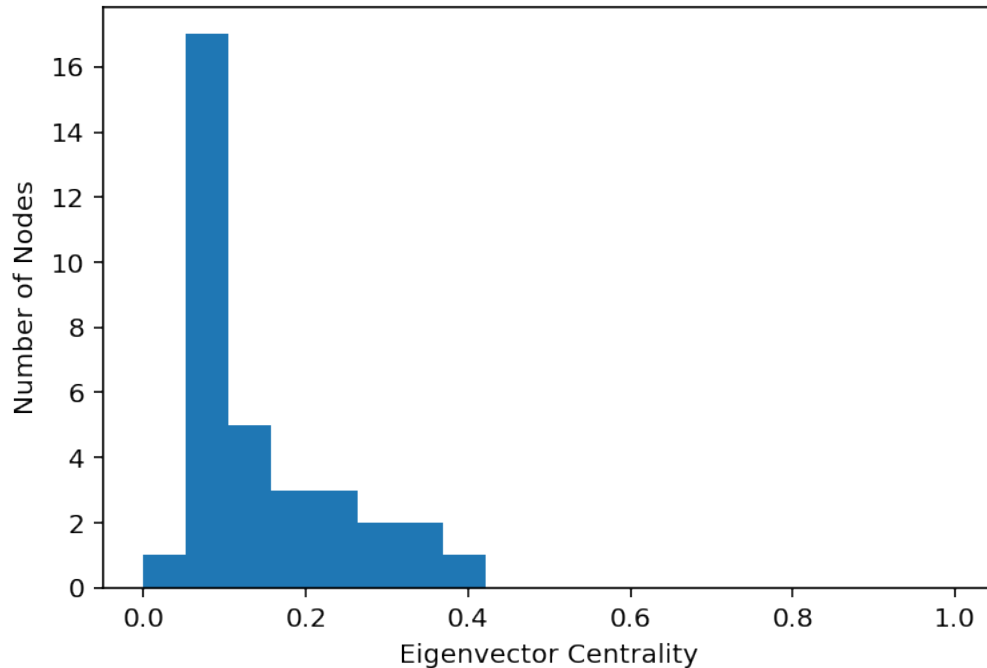
Let's check them out on our test case, the karate club:

```
In [78]: # plotting the graph
cent = np.array(list(nx.eigenvector_centrality(Gk).values()))
scaler = MinMaxScaler((50,800))
scaledC = scaler.fit_transform(cent[:,np.newaxis])
nx.draw_networkx(Gk, node_size=scaledC, pos=pos, with_labels=False, node_color='#6699cc')
plt.axis('off');
```



Notice how the large nodes form an "inner circle" of this social network.

```
In [79]: plt.hist(cent,bins=np.linspace(0,1,20))
plt.xlabel('Eigenvector Centrality')
_=plt.ylabel('Number of Nodes')
```



For this network, the eigenvector centrality of nodes is more evenly distributed than the betweenness centrality.

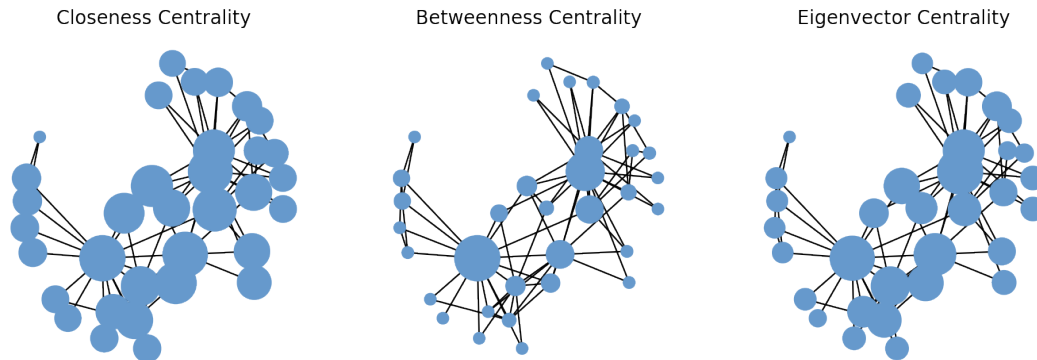
```
In [80]: plt.figure(figsize=(12,4))
plt.subplot(131)
cent = np.array(list(nx.closeness centrality(Gk).values()))
scaler = MinMaxScaler((50,800))
scaledC = scaler.fit_transform(cent[:,np.newaxis])
nx.draw_networkx(Gk, node_size=scaledC, pos=pos, with_labels=False, node_color='#6699cc')
plt.title("Closeness Centrality")
plt.axis('off')

plt.subplot(132)
cent = np.array(list(nx.betweenness centrality(Gk).values()))
scaler = MinMaxScaler((50,800))
scaledC = scaler.fit_transform(cent[:,np.newaxis])
nx.draw_networkx(Gk, node_size=scaledC, pos=pos, with_labels=False, node_color='#6699cc')
plt.title("Betweenness Centrality")
plt.axis('off')

plt.subplot(133)
cent = np.array(list(nx.eigenvector centrality(Gk).values()))
scaler = MinMaxScaler((50,800))
scaledC = scaler.fit_transform(cent[:,np.newaxis])
nx.draw_networkx(Gk, node_size=scaledC, pos=pos, with_labels=False, node_color='#6699cc')
```



```
_=plt.title("Eigenvector Centrality")
plt.axis('off');
```



Pagerank Centrality.

So far, the measures we have been talking about could apply to either directed or undirected graphs.

When one focuses on directed graphs, some new ideas can be applied.

An important example concerns the Web: a link is a **directed** edge from one Web page to another.

To set the stage for understanding Pagerank:

In the mid to late 1990's, there were literally dozens of search engines for the Web.

At present, most of them are no longer in existence, because Google emerged in 1998 and came to dominate Web search almost overnight.

How did this happen?

As background: a typical search engine uses a two-step process to retrieve pages related to a user's query.

In the first step, basic text processing is done to find all documents that contain the query terms. Due to the massive size of the Web, this first step can result in many thousands of retrieved pages related to the query.

Some of these pages are important, but most are not.

The problem that Google solves better than the search engines of the mid 1990's concerns the **ordering** in which the resulting search results are presented. This is the crucial factor in utility. A user wants to find the best item at the top of the search results.

By displaying the most relevant pages at the top of the list returned each query, Google makes its search results very useful. The algorithm that gave Google this advantage is called Pagerank.

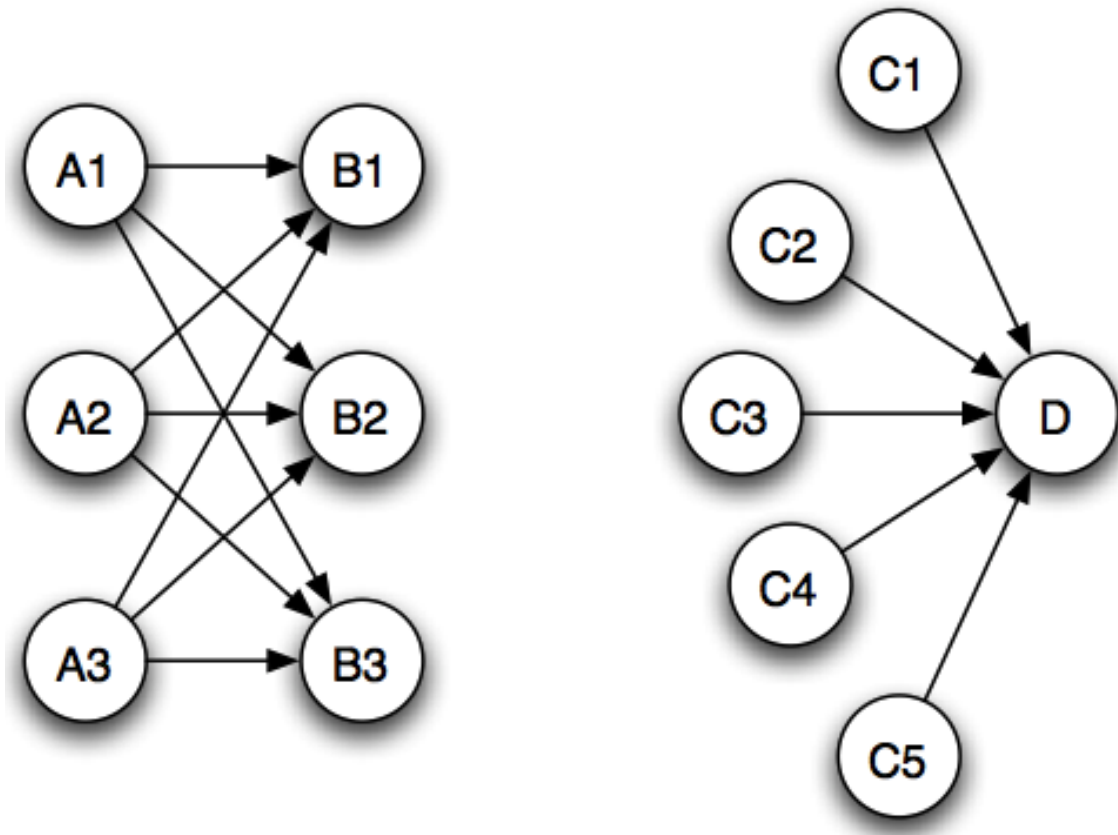
The Insight

Around 1998, the limitations of standard search engines, which just used term frequency, were becoming apparent. A number of researchers were thinking about using additional sources of information to "rate" pages.

The key idea that a number of researchers hit on was this: *links are endorsements*.

When a first page contains a link to a second page, that is an indication that the author of the first page thinks the second page is worth looking at. If the first and second pages both contain the same query terms, it is likely that the second page is an important page with respect to that query term.

Consider a set of web pages, for a single query term (say "car manufacturers") with this linking structure:



It may be clear that the links between pages contain useful information. But what is the best way to extract that information in the form of rankings?

Here is the strategy that Brin and Page used:

From *"The PageRank citation ranking: Bringing order to the Web"* (1998): > 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. The probability that the random surfer visits a page is its PageRank.

1.4 Random Walks

We start with the notion of a **random walk**.

A random walk is a model of many sorts of processes that occur on graphs.

Let us fix a graph G . A random walk (sometimes called a "drunkard's walk") models the movement of an object on this graph.

We assume that the object moves from node to node in G , one move per time step t . At time t the object is at node k (say) and at the next time $t + 1$ it moves to another node chosen **at random** from among the outgoing edges of k .

We can capture the process of movement on G as a **Markov chain**.



Andrei Markov, 1856 - 1922, St Petersburg.

Markov was part of the great tradition of mathematics in Russia, which carried into and beyond the Soviet era. Among other pursuits, Markov pioneered the study of systems in which the future state of the system depends only on the present state in a random fashion. Classic examples include the movement of stock prices and the dynamics of animal populations.

These have since been termed "Markov Chains."

A Markov chain involves three definitions:

Definition. A *probability vector* is a vector of nonnegative entries that sums to 1.

Definition. A *stochastic matrix* is a square matrix of nonnegative values whose columns each sum to 1.

Definition. A *Markov chain* is a system whose state is a probability vector and which evolves according to a stochastic matrix.

That is, it is a probability vector \mathbf{x}_0 and a stochastic matrix $P \in \mathbb{R}^{n \times n}$ such that

$$\mathbf{x}_{k+1} = P\mathbf{x}_k \quad \text{for } k = 0, 1, 2, \dots$$

So we think of a probability vector \mathbf{x}_0 as describing how things are "distributed" across various categories -- the fraction of items that are in each category.

And we think of the stochastic matrix P as describing how things "redistribute" themselves at each time step.

A important question about a Markov Chain is: what will happen in the distant future?

In particular we are often concerned with the **steady state** of a Markov Chain.

This is some vector \mathbf{x} which is the limiting value of $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots$

The way to interpret the steady-state of the Markov chain in terms of the random walk is:

Let the chain (random walk) start in the given state. At some long time in the future, make an observation of the state that the chain is in. Then the steady-state distribution gives, for each state, the probability that the chain is in that state when we make our observation.

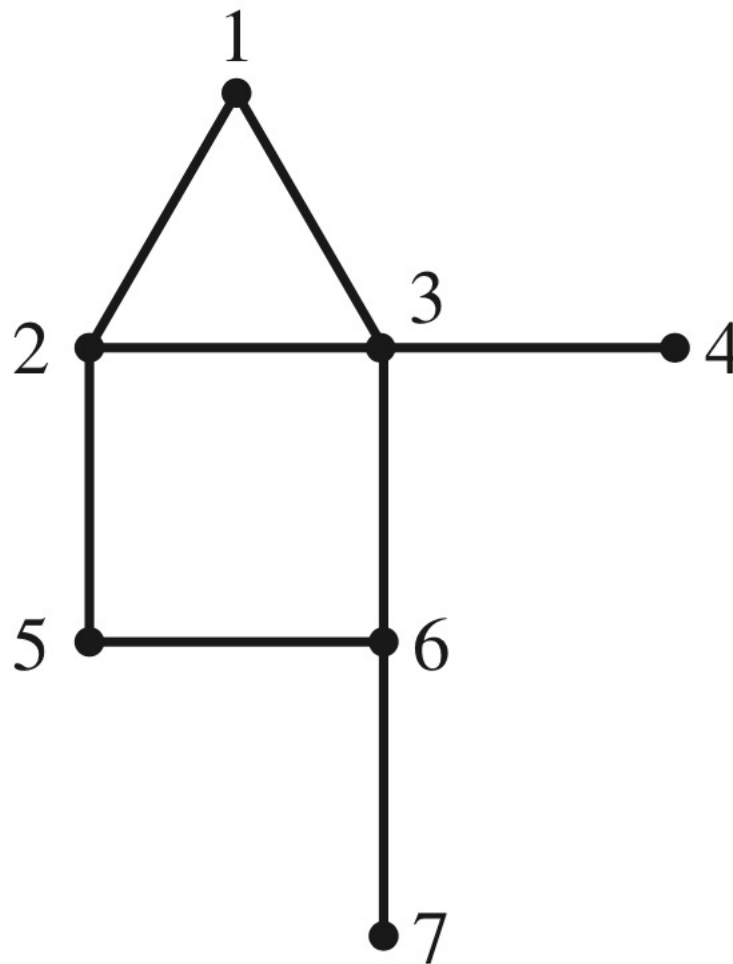
When does a Markov Chain have a steady state, and how can we find it?

Here are important facts about a Markov Chain.

For a Markov Chain having transition matrix P :

- The largest eigenvalue of P is 1.
- If P is regular, then
 - There is only one eigenvalue equal to 1
 - The chain will converge to the corresponding eigenvector as its *unique steady-state*.
- " P is regular" means that for some $k > 0$, all entries in P^k are nonzero.

Now let's consider a random walk on an undirected graph:



Again, at each node there is an equal probability of departing to any adjacent node. The transition matrix associated with a random walk on this graph is

$$P = \begin{bmatrix} 0 & 1/3 & 1/4 & 0 & 0 & 0 & 0 \\ 1/2 & 0 & 1/4 & 0 & 1/2 & 0 & 0 \\ 1/2 & 1/3 & 0 & 1 & 0 & 1/3 & 0 \\ 0 & 0 & 1/4 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 \\ 0 & 0 & 1/4 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1/3 & 0 \end{bmatrix}$$

It turns out that this matrix is regular (P^3 has no zero entries.)

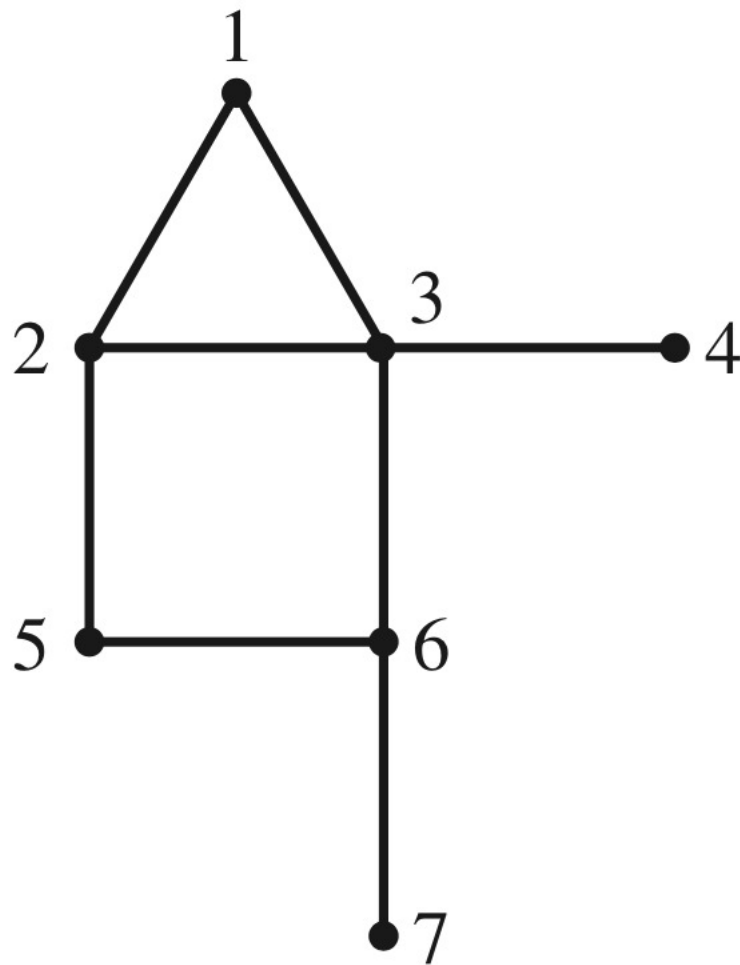
Hence, the associated Markov Chain converges to a single steady state. (It has only one eigenvalue of 1.)

We can obtain the corresponding eigenvector via standard numerical libraries (eg, `np.linalg.eig()`). The eigenvector corresponding to the eigenvalue of 1 is the steady-state of the Markov Chain.

Hence we can find that the steady-state is $\frac{1}{16} \begin{bmatrix} 2 \\ 3 \\ 4 \\ 1 \\ 2 \\ 3 \\ 1 \end{bmatrix}$.

That is, the probability of being in node 1 at steady state is 2/16; the probability of being in node 2 is 3/16; the probability of being in node 3 is 4/16, etc.

Notice anything? (Look at G again).

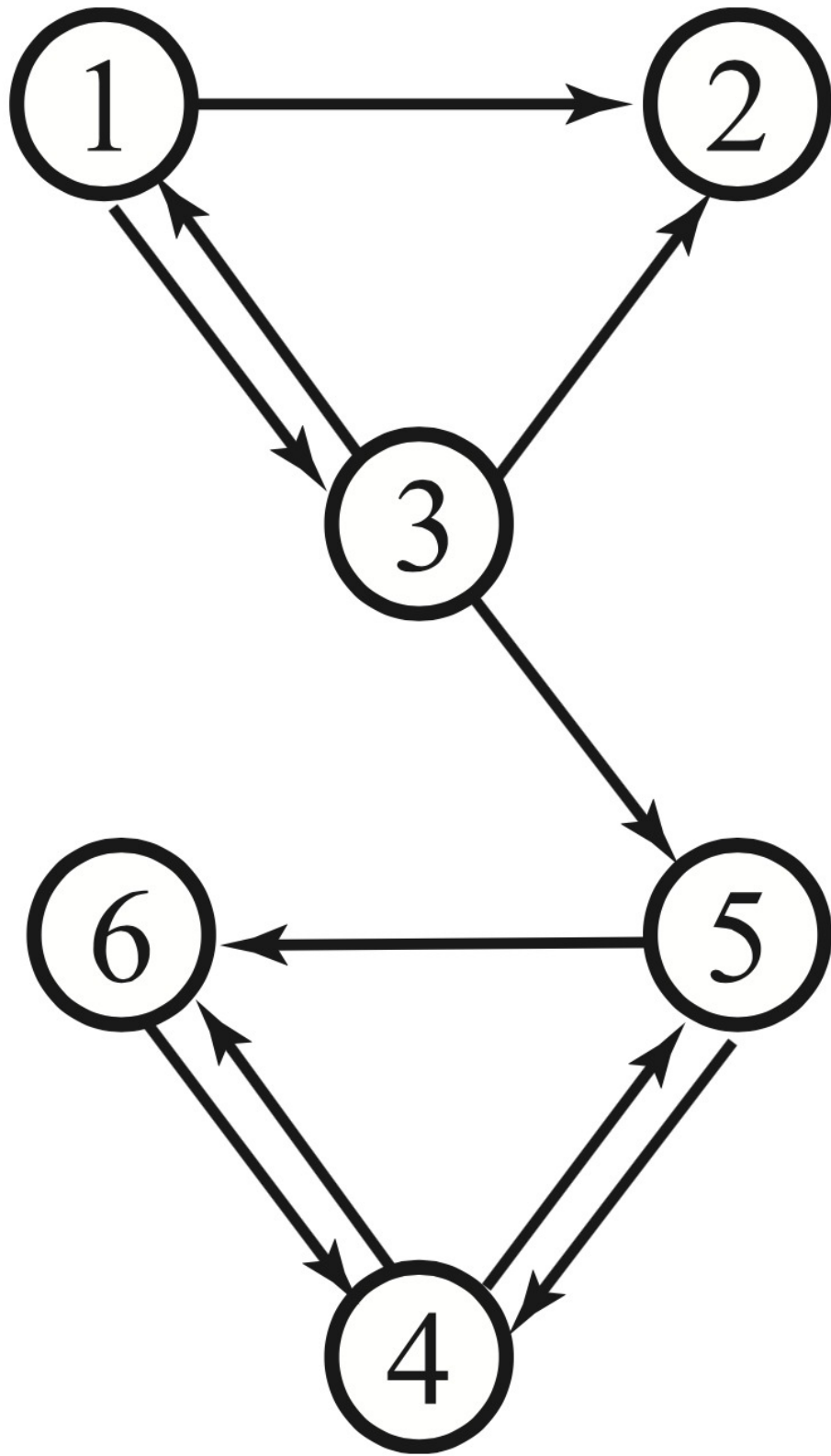


It can be proved that the steady-state distribution of a random walk on an **undirected** graph is proportional to node degree.

That is, the probability of being at a particular node at steady state is proportion to that node's degree.

A more interesting object on which to walk randomly is a **directed** graph. In this graph, all edges are "one-way streets" -- nodes are joined not by lines but by arrows. The chain can move from vertex to vertex, but only in the directions allowed by the arrows.

An example of a directed graph is



The transition matrix for this graph is:

$$P = \begin{bmatrix} 0 & 0 & 1/3 & 0 & 0 & 0 \\ 1/2 & 0 & 1/3 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1 \\ 0 & 0 & 1/3 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{bmatrix}$$

We can conclude that this matrix is **not** regular. Why?

One reason we can conclude this is the column of zeros (column 2). Any power of A will preserve this column of zeros.

1.5 PageRank

There are many ways to make use of the link structure to infer which pages are most important to return at the top of the search results. (There was a lot of experimentation in the late 1990s with various methods).

A simple method is just to consider a page is "important" if many "important" pages link to it. More precisely, this definition of "importance" is:

$$\text{Importance of page } k = \sum_j (\text{Importance of page } j) \cdot (\text{Probability of going from page } j \text{ to page } k.)$$

This can be captured in terms of a random walk.

Now we are ready to understand what Page and Brin were saying in 1998:

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. The probability that the random surfer visits a page is its PageRank.

Intuitively, the a random surfer should spend more time at "important" pages and less time at unimportant pages.

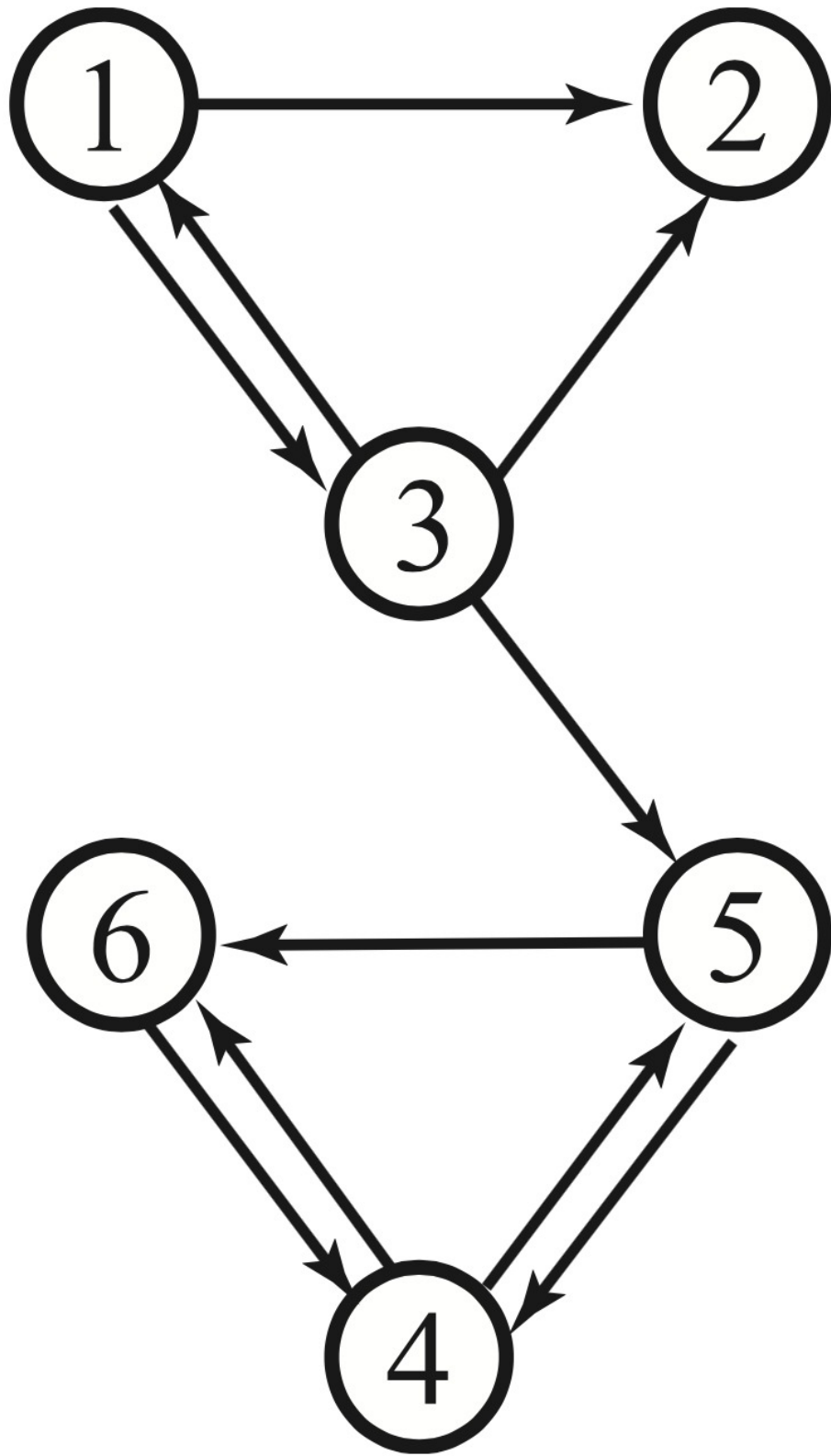
The way to interpret this precisely is:

- 1) Form the graph that encodes the connections between Web pages that are retrieved for a particular query.
- 2) Construct a Markov chain that corresponds to a random walk on this graph.
- 3) Rank-order the pages according to their probability in the Markov chain's steady state.

So let's try to make this work and see what happens.

Example. Assume a set of Web pages have been selected based on a text query, eg, pages related to "personal 737 jets."

These pages have various links between them, as represented by this graph:



Let us try to construct the unique steady-state distribution for a random walk on this graph, if it exists. That is, we will construct the PageRank for this set of Web pages.

Solution.

The key question we must ask is **whether a unique steady state exists.**

Step 1.

Assume there are n pages to be ranked. Construct an $n \times n$ transition matrix for the Markov chain.

Set the Markov chain transitions so that each outgoing link from a node has equal probability of being taken.

We have already seen the transition matrix for this graph:

$$P = \begin{bmatrix} 0 & 0 & 1/3 & 0 & 0 & 0 \\ 1/2 & 0 & 1/3 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1 \\ 0 & 0 & 1/3 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{bmatrix}$$

We have observed that this transition matrix is **not** regular, because for any $P^k, k > 0$, the second column will be zero.

To address this, let's ask why it happens.

The reason that column 2 of P is zero is that the Web page corresponding to node 2 has no links embedded in it, so there is nowhere to go from this page. Of course this will happen a lot in an arbitrary collection of Web pages.

Note that Page and Brin say that the random surfer will occasionally "start on another random page." In other words, it seems reasonable that when reaching a page with no embedded links, the surfer chooses another page at random.

So this motivates the first adjustment to P :

Step 2:

Form the matrix P' as follows: for each column in P that is entirely zeros, replace it with a column in which each entry is $1/n$.

In our example:

$$P = \begin{bmatrix} 0 & 0 & 1/3 & 0 & 0 & 0 \\ 1/2 & 0 & 1/3 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1 \\ 0 & 0 & 1/3 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{bmatrix} \rightarrow P' = \begin{bmatrix} 0 & 1/n & 1/3 & 0 & 0 & 0 \\ 1/2 & 1/n & 1/3 & 0 & 0 & 0 \\ 1/2 & 1/n & 0 & 0 & 0 & 0 \\ 0 & 1/n & 0 & 0 & 1/2 & 1 \\ 0 & 1/n & 1/3 & 1/2 & 0 & 0 \\ 0 & 1/n & 0 & 1/2 & 1/2 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1/6 & 1/3 & 0 \\ 1/2 & 1/6 & 1/3 & 0 \\ 1/2 & 1/6 & 0 & 0 \\ 0 & 1/6 & 0 & 0 \\ 0 & 1/6 & 1/3 & 1/2 \\ 0 & 1/6 & 0 & 1/2 \end{bmatrix}$$

Nonetheless, even after this change, P' can fail to be regular.

In other words, for an arbitrary set of web pages, there is no guarantee that their transition matrix will be regular.

Once again, let's read the words of Page and Brin closely: the surfer "eventually gets bored and starts on another random page."

Step 3.

In practice this means that there a small probability that the surfer will jump from any page to any other page at random.

Let's call this small probability α .

We can't just add α to every entry in P' , because then the columns of the new matrix would not sum to 1.

Instead we decrease each entry in P' by a factor of $(1 - \alpha)$, and then add α/n to it.

So we compute the final transition matrix P'' as:

$$P''_{ij} = (1 - \alpha)P'_{ij} + \frac{\alpha}{n}.$$

We can write this as a matrix equation:

$$P'' = (1 - \alpha)P' + \frac{\alpha}{n}\mathbf{1}$$

where $\mathbf{1}$ is an $n \times n$ matrix of 1's.

In our example, let's say that $\alpha = 1/10$ (in reality it would be smaller). So $\alpha/n = 1/60$.

Then:

$$P' \begin{bmatrix} 0 & 1/6 & 1/3 & 0 & 0 & 0 \\ 1/2 & 1/6 & 1/3 & 0 & 0 & 0 \\ 1/2 & 1/6 & 0 & 0 & 0 & 0 \\ 0 & 1/6 & 0 & 0 & 1/2 & 1 \\ 0 & 1/6 & 1/3 & 1/2 & 0 & 0 \\ 0 & 1/6 & 0 & 1/2 & 1/2 & 0 \end{bmatrix} \rightarrow P'' = \begin{bmatrix} 1/60 & 1/6 & 19/60 & 1/60 & 1/60 & 1/60 \\ 7/15 & 1/6 & 19/60 & 1/60 & 1/60 & 1/60 \\ 7/15 & 1/6 & 1/60 & 1/60 & 1/60 & 1/60 \\ 1/60 & 1/6 & 1/60 & 1/60 & 7/15 & 11/12 \\ 1/60 & 1/6 & 19/60 & 7/15 & 1/60 & 1/60 \\ 1/60 & 1/6 & 1/60 & 7/15 & 7/15 & 1/60 \end{bmatrix}$$

Obviously, P'' is regular, because all its entries are positive (they are at least α/n .)

P'' is the Markov Chain that Brin and Page defined, and which is used by PageRank to rank pages in response to a Google search.

Step 4. Compute the steady-state of P'' , and rank pages according to their magnitude in the resulting vector.

We can do this by solving $P''\mathbf{x} = \mathbf{x}$, or we can compute the eigenvectors of P'' and use the eigenvector that corresponds to $\lambda = 1$.

For the example P'' , we find that the steady-state vector is:

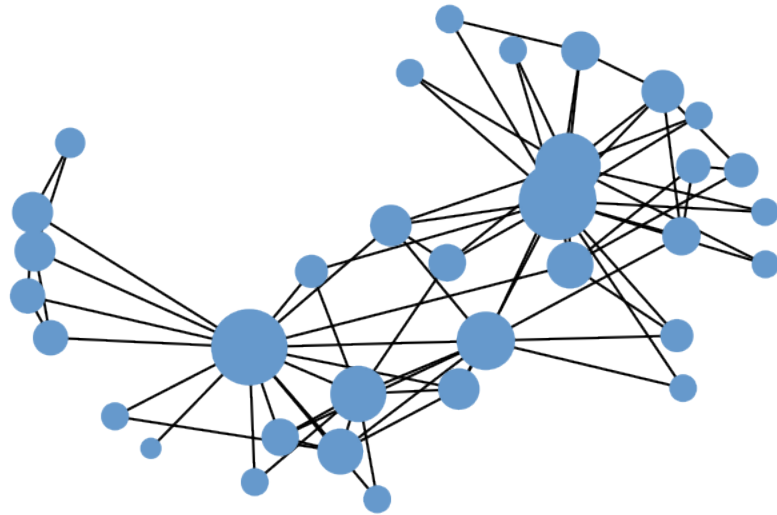
$$\mathbf{x} = \begin{bmatrix} 0.071 \\ 0.104 \\ 0.080 \\ 0.720 \\ 0.395 \\ 0.549 \end{bmatrix}$$

So the final ranking of pages is: 4, 6, 5, 2, 3, 1.

This is the order that PageRank would display its results, with page 4 at the top of the list.

Returning to the karate club graph:

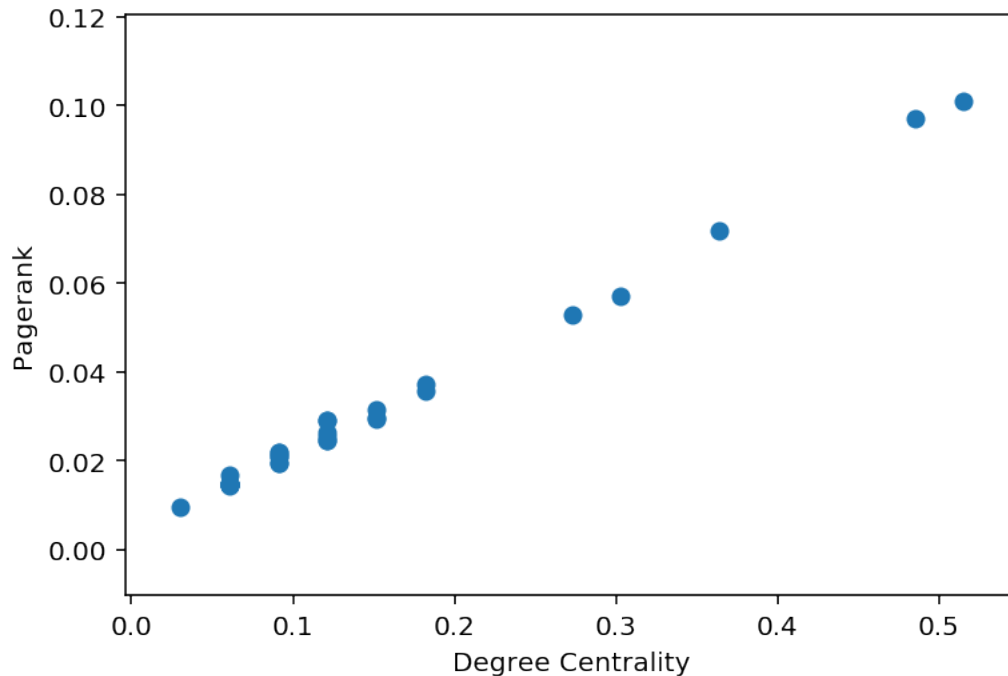
```
In [89]: # plotting the graph
cent = np.array(list(nx.pagerank(Gk).values()))
scaler = MinMaxScaler((50,800))
scaledC = scaler.fit_transform(cent[:,np.newaxis])
nx.draw_networkx(Gk, node_size=scaledC, with_labels=False, pos=pos, node_color='#6699cc')
plt.axis('off');
```



Because this graph is undirected, the pagerank of a node is just proportional to its degree.

```
In [90]: degree_c = np.array(list(nx.degree_centrality(Gk).values()))  
plt.scatter(degree_c, cent)  
plt.xlabel('Degree Centrality')  
plt.ylabel('Pagerank')
```

```
Out[90]: <matplotlib.text.Text at 0x11a15bdd8>
```



To study a more interesting directed graph, we can look at the nervous system of the flatworm (*C. elegans*):

```
In [91]: G = nx.read_gml('data/celegansneural.gml')
scc = nx.strongly_connected_component_subgraphs(G)
sizes = [(g, len(g)) for g in scc]
largestG = max(sizes, key = lambda x: x[1])[0]
```

```
In [92]: Gmax = nx.DiGraph(largestG)
degree_c = nx.degree_centrality(Gmax)
pagerank_c = nx.pagerank(Gmax)
eigenvector_c = nx.eigenvector_centrality(Gmax)
betweenness_c = nx.betweenness_centrality(Gmax)
```

```
In [93]: n = len(Gmax.nodes())
deg = np.zeros(n)
pr = np.zeros(n)
eig = np.zeros(n)
bw = np.zeros(n)
i=0
for node in Gmax:
    deg[i] = degree_c[node]
    pr[i] = pagerank_c[node]
    eig[i] = eigenvector_c[node]
    bw[i] = betweenness_c[node]
    i+=1
```

```

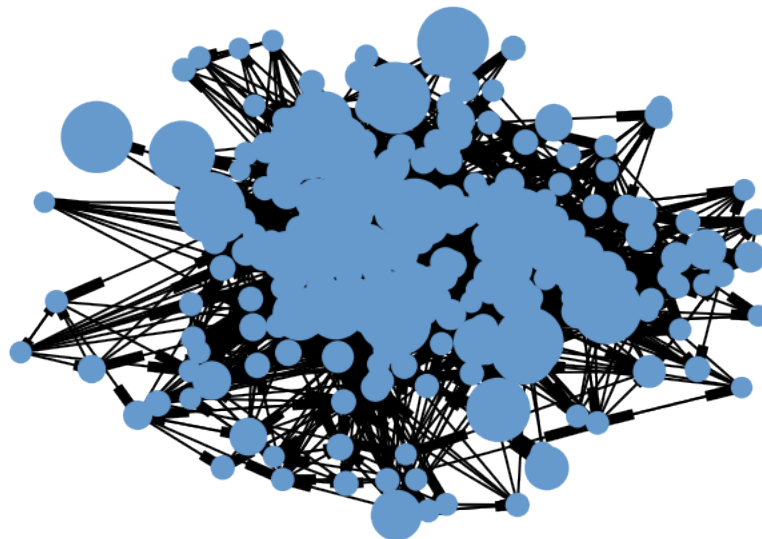
measures = pd.DataFrame()
measures['nodes'] = Gmax.nodes()
measures.set_index(['nodes'], inplace=True)
measures['eigenvector_c'] = pd.DataFrame.from_dict(eigenvector_c, orient='index')
measures['pagerank_c'] = pd.DataFrame.from_dict(pagerank_c, orient='index')
measures['degree_c'] = pd.DataFrame.from_dict(degree_c, orient='index')
measures['betweenness_c'] = pd.DataFrame.from_dict(betweenness_c, orient='index')

```

```

In [94]: # plotting the graph
cent = np.array(list(nx.pagerank(Gmax).values()))
scaler = MinMaxScaler((50,800))
scaledC = scaler.fit_transform(cent[:,np.newaxis])
nx.draw_networkx(Gmax, node_size=scaledC, with_labels=False, node_color='#6699cc')
plt.axis('off');

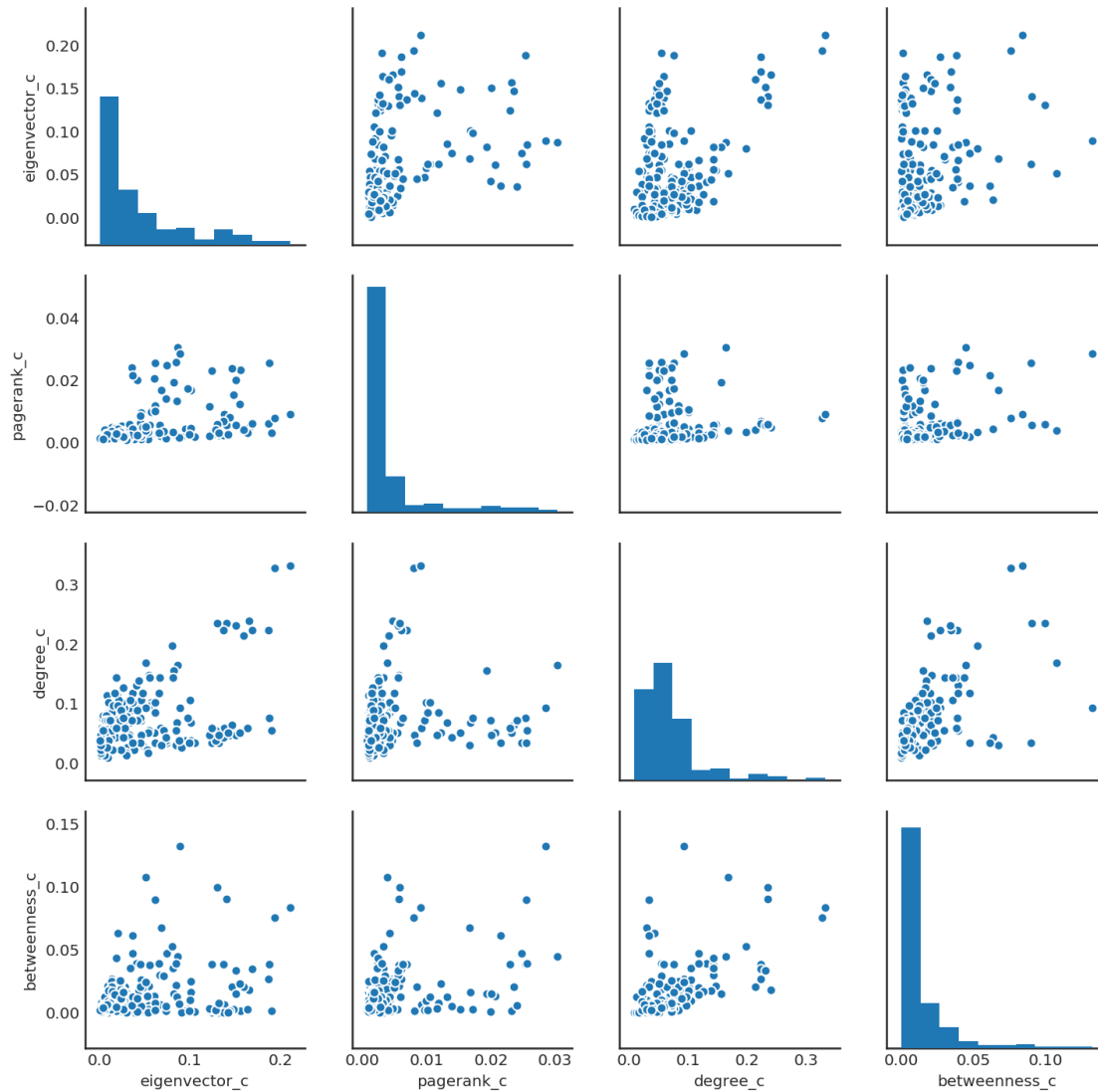
```



```

In [95]: with sns.axes_style('white'):
sns.pairplot(measures)

```



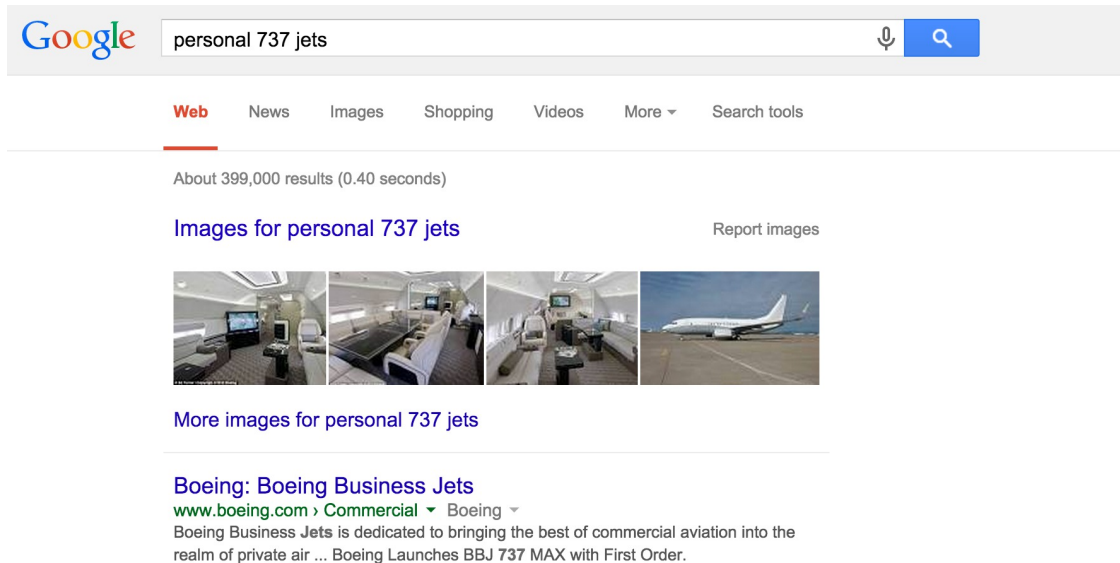
1.6 Computing PageRank: the Power Method

From a mathematical standpoint, we are done!

However, from a Computer Science standpoint, there are still some issues. :)

The most significant issue is simply this: PageRank results must be provided **very quickly**. Search engines are in competition and speed is a competitive advantage.

Here is an example Google search:



Notice that the search returned about 400,000 results!

Recall that using Gaussian elimination to solve $A\mathbf{x} = \mathbf{b}$ takes about $\frac{2}{3}n^3$ operations.

In this case, apparently $n = 400,000$.

So computing the PageRank in the straightforward way we've described would take about 42,667,000,000,000,000 operations.

Assuming a 2GHz CPU, that's on the order of **eight months**.

In [97]: $((2./3)*(400000**3))/((2*10**9)*(3600*24*30))$

Out [97]: 8.23045267489712

We need a faster way to compute the PageRank!

Here is an important point: we only need the **principal** eigenvector. (The one corresponding to $\lambda = 1$).

Let's review how a Markov chain gets to steady state. As we discussed at the end of the lecture on the characteristic equation, the state of the chain at any step k is given by

$$\mathbf{x}_k = c_1 \mathbf{v}_1 \lambda_1^k + c_2 \mathbf{v}_2 \lambda_2^k + \cdots + c_n \mathbf{v}_n \lambda_n^k.$$

Let's assume that λ_1 is the eigenvalue 1. If the chain converges to steady state, then we know that all eigenvalues other than λ_1 are less than 1 in magnitude.

Of course, if $|\lambda_i| < 1$,

$$\lim_{k \rightarrow \infty} \lambda_i^k = 0.$$

So:

$$\lim_{k \rightarrow \infty} \mathbf{x}_k = c_1 \mathbf{v}_1.$$

Note that c_1 is just a constant that doesn't affect the relative sizes of the components of \mathbf{x}_k in the limit of large k .

This is another way of stating that the Markov chain goes to steady state **no matter what the starting state is**.

This observation suggests another way to compute the steady state of the chain:

1. Start from a **random** state \mathbf{x}_0 .
2. Compute $\mathbf{x}_{k+1} = A\mathbf{x}_k$ for $k = 0, 1, 2, 3, \dots$

How do we know when to stop in Step 2?

Since we are looking for steady-state, we can stop when the difference between \mathbf{x}_{k+1} and \mathbf{x}_k is small.

This is called the **power method**.

Why is this a better method?

Keep in mind that the number of flops in matrix-vector multiplication is about $2n^2$.

This is compared to $2/3n^3$ for solving a system (finding the eigenvector directly).

Let's say that after computing

$$\mathbf{x}_1 = A\mathbf{x}_0$$

$$\mathbf{x}_2 = A\mathbf{x}_1$$

$$\mathbf{x}_3 = A\mathbf{x}_2$$

$$\mathbf{x}_4 = A\mathbf{x}_3$$

$$\mathbf{x}_5 = A\mathbf{x}_4$$

$$\mathbf{x}_6 = A\mathbf{x}_5$$

$$\mathbf{x}_7 = A\mathbf{x}_6$$

$$\mathbf{x}_8 = A\mathbf{x}_7$$

$$\mathbf{x}_9 = A\mathbf{x}_8$$

$$\mathbf{x}_{10} = A\mathbf{x}_9$$

we find that \mathbf{x}_{10} is sufficiently close to \mathbf{x}_9 .

How much work did we do?

We did 10 matrix-vector multiplications, or $20n^2$ flops.

So the power method is

$$\frac{\frac{2}{3}n^3}{20n^2} = \frac{n}{30}$$

times faster than the direct method.

For our example, $n/30 = 13,333$. So this trick reduces the running time from **8 months** down to **27 minutes**.

```
In [98]: 20*400000.**2/((2*10**9)*(60))
```

```
Out[98]: 26.666666666666668
```

```
In [99]: # Given time, talk about sparse matrix-vector multiply.  
# This is linear in n, with a constant equal to average degree (say 10)  
# and gets the computation down to 40 milliseconds  
# but that requires explaining why P is not really dense  
# (it can be expressed as a sparse matrix plus a constant matrix)  
20*10*400000./((2*10**9))
```

Out [99] : 0.04

This is an example of an **iterative** method. Iterative methods are often the preferred approach for solving linear algebra problems in the real world.

One final thing: how exactly do we decide when to stop iterating in the power method?

One simple way is to add up the differences of the components of $\mathbf{x}_{k+1} - \mathbf{x}_k$:

$$s = \sum_{i=1}^n |\mathbf{x}_{k+1,i} - \mathbf{x}_{k,i}|$$

and compare it to the sum of the components of \mathbf{x}_k :

$$d = \sum_{i=1}^n |\mathbf{x}_{k,i}|$$

If s/d is small (say, less than 0.001) then we can conclude that \mathbf{x}_{k+1} is close enough to \mathbf{x}_k for us to stop iterating.

So the power method is fast, making it the algorithm of choice for a company like Google. It is also easy to implement, and easy to parallelize across multiple machines.