LAB 10 - NETWORKS AND EIGENVECTOR CENTRALITY

1. Introduction

Last week we learned how to use iterative techniques to find the dominant eigenvector for a system, and used this to analyze a Markov process. This week we are going to further explore these techniques and see how they can be used to analyze networks.

Simply put, a network consists of a collection of *nodes* and *edges*, where each edge can be thought of as a connection which joins two nodes. We often represent networks graphically, by drawing a dot for each node, and a line for each edge as shown below in Figure 1. Networks are sometimes referred to as *graphs* in mathematics, and the nodes are sometimes referred to as *vertices*.

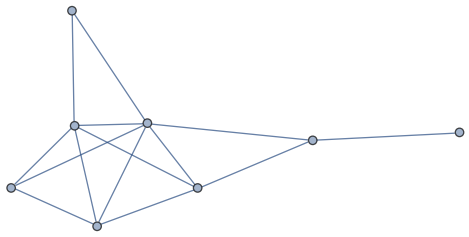

Figure 1. A network with 8 nodes (dots), and 14 edges (lines).

Networks are incredibly useful in real world applications, as they allow us to represent relationships between objects in a wide variety of different systems. For example, a data scientist might use networks to represent connections on social media, by displaying each Facebook user as a node of the network, with an edge joining two nodes when the corresponding users are Facebook friends (see Figure 2). Urban planners may use networks to design important infrastructure, by representing water pump station as nodes, and water mains as edges. Law enforcement officials may use networks to understand criminal organizations, with gang members represented by nodes, and known affiliations between members represented by edges. Finally, we could use networks to represent Hollywood actors and actresses, connecting two actors/actresses if they've starred in a movie together. This would allow us to answer definitively whether all of Hollywood really is 6 degrees from Kevin Bacon.

By introducing a slight variation on the idea of a network as defined above, we can capture other types of relationships. For example, we could add a direction to each of our edges, and think of them as providing a connection *from* one node, *to* another node. We indicate the direction of each edge by adding an arrow, and call the resulting network a *directed network* (see Figure 5).

Directed networks allow us to model relationships that are not symmetric. For example, the food chain could be represented as a directed network, with each node representing a different species, and a directed edge added which travels from each predator to the prey it is known to eat. In this network there would be an edge from the node representing lions to the node representing gazelles, because lions are known to eat gazelles, but not one going the other way.

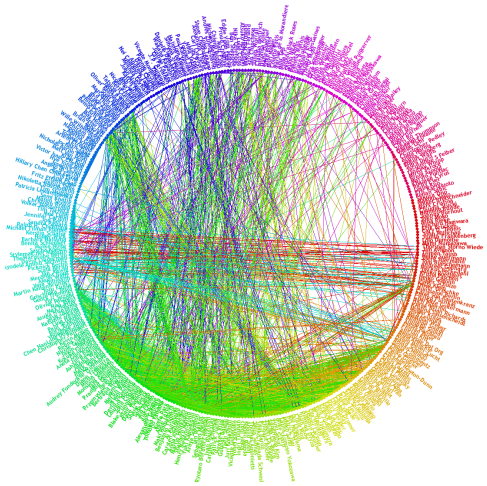

FIGURE 2. A Facebook "Friend Wheel", where each of a single user's friends is represented by a node on the edge of the circle, with a colorful edge joining two friends precisely when they are Facebook friends with one another.

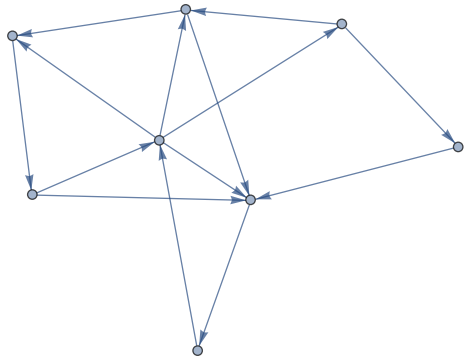

FIGURE 3. A directed network with 8 nodes (dots), and 14 directed edges (arrows).

2. The World Wide Web

Perhaps one of the most ubiquitous directed networks in our everyday world is the internet, or the world wide web. Understanding this network is crucial to a number of tasks that we do everyday, often without thinking twice about them. It is this network that we will focus on in this lab.

There are several different ways in which the internet can be modeled as a network, each of which captures a different aspect of the web. For example, we could model the physical infrastructure of the internet (wires, hubs, and internet service providers) as a network, to model the way in which data is sent from one computer to another over the internet. Another way to represent the internet is by assigning a node to each of the estimated 1.8 billion webpages on the internet, and then adding a directed edge from webpage A to webpage B when there is a link on webpage A which redirects the user to webpage B. It is this second viewpoint which we will be using in this lab.

To understand the importance of being able to study this network, consider the problem of designing an internet search engine, like *Google*, *Yahoo*, *Bing*, or *AltaVista*. A user comes to

the search engine website and types in a search term, say "Spider-Puppy", hoping to find a website with an image like the one below:

FIGURE 4. The intended goal of the user's websearch.

Long before the user decided to search for these terms, the search engine created a catalogue with all (actually in practice, only many) of the websites on the internet, as well as the searchable terms they contain. At the time the user performs his/her websearch, the search engine creates a list of all of the websites in its catalogue which contain the word "Spider-Puppy". Before displaying the results of your search, there is one important question the search engine needs to answer. *Which of the search results should it display first, at the top of the search page?*

Search engine users expect that the results which are most relevant to their search should be displayed at, or near, the top of the search page. But how does the search engine decide this?

For simplicity, suppose that our search engine finds 8 webpages containing the word "Spider-Puppy". These webpages each contain links to each other, as displayed in the directed network below.

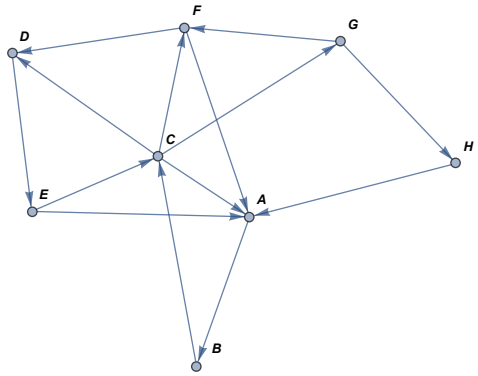

FIGURE 5. Websites containing our desired search terms. How do we decide which webpage is the most important?

We would like to find a way to decide which webpage is the most relevant to the user, or in other words, which website should be considered the most important. One obvious way to do this would be to pick the nodes (i.e. webpages) that have the most incoming edges (i.e. links from other pages). Using this criteria, we would expect in our above network that webpage A should be the most important, because it has the most incoming links from other webpages (4

in total). On the other hand, webpages with only one incoming link like B, E, G, and H, will be unimportant, and should therefore be listed towards the bottom of the search results.

Notice, however, that even though webpage B only has one other webpage linking to it, that linking webpage happens to be the important webpage A. In fact, the page B is the *only* page that A links to. So it is reasonable to expect that this indicates that page B is important as well, even though it only has one incoming link. *Is there a way to assign rankings to the search results, which not only take into account the number of links coming into a given webpage, but also the relative importance of those links?*

Of course, with only 8 webpages we could probably come up with a reasonable ranking by hand. The problem is that in most situations the number of webpages containing the desired search terms is in the thousands, or even millions. In what follows we will see a simplified version of one algorithm, called *PageRank*, which answers this question. PageRank was designed by Sergey Brin and Larry Page, who shortly afterwards founded Google, using the PageRank algorithm in their search engine. Today PageRank remains one of the core components in how Google sorts and ranks search results.

3. Adjacency Matrices

In this lab we will be working with a network which has 499 nodes, each representing a webpage, and 12650 directed edges between them representing links between the different webpages. Each node (webpage) is labelled with an integer from 0 to 498, while the directed edges (links) are represented in the file `Lab10webpagedata.csv` as a 12650×2 matrix,

$$\begin{bmatrix} 76 & 109 \\ 76 & 4 \\ 76 & 78 \\ \vdots & \vdots \end{bmatrix}.$$

Here we interpret the first row as telling us that there is a link from webpage number 76 to webpage number 109, while the second row tells us there is a link from webpage number 76 to webpage number 4, etc.

- Using the instructions and code provided in the lab notebook, import the data in `Lab10webpagedata.csv` and use it to create a NumPy array called `webpage_data`.
- Using the code in the notebook, create a visual representation of the network.

There is a more useful way of representing this network as a matrix (instead of representing it as a matrix with a row for every edge), which we call the *adjacency* matrix of the network. Suppose that our network G has n nodes. Then the adjacency matrix of the network G is an $n \times n$ matrix, where each column and each row correspond to one of the nodes of a network. For example, if our network has nodes $v_1, v_2, \ldots, v_n$, then the first row corresponds to v_1, the second row corresponds to v_2, etc. Likewise, the first column corresponds to v_1, the second column corresponds to v_2, and so on. Then the entry in the (i, j)th position of our adjacency matrix (ith row and jth column) will be the number of edges from v_i to v_j.

Consider for example the network G with 5 nodes shown in Figure 6. It's 10 edges can be described by the matrix

$$E = \begin{bmatrix} 0 & 1 \\ 1 & 4 \\ 0 & 2 \\ 0 & 4 \\ 1 & 3 \\ 2 & 0 \\ 2 & 4 \\ 3 & 4 \\ 3 & 2 \\ 3 & 2 \end{bmatrix}.$$

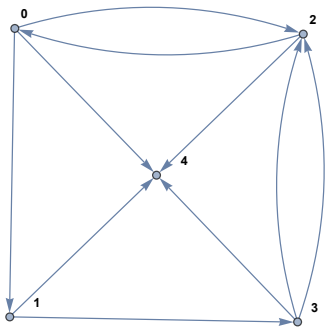

FIGURE 6. A directed network G.

We can instead represent this information by a 5×5 adjacency matrix, which is given by

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Notice the 2 in the 4th row, 3rd column, since there are two edges traveling from vertex 3 to vertex 2. Also, notice that there are no nonzero entries in the last row, which corresponds to the fact that vertex 4 does not have any edges which start from it.

Problem 1. Write a function `adj_matrix(n,edge_list)` which accepts as input a number of vertices `n` and a list of edges, `edge_list` (which is formatted as a $m \times 2$ NumPy array, where m is the number of edges), and constructs the `n` $\times$ `n` adjacency matrix `adj_m`. The output `adj_m` should be a NumPy array. You may assume that the vertices of the graph referred to in `edge_list` are labelled `0`, `1`, $\cdots$, `n-1`.

Hint: Create an `n` $\times$ `n` *matrix of all zeros first using the command* `np.zeros((n,n))`, *and then loop through the list* `edge_list` *to correctly place the nonzero entries in the adjacency matrix. Remember, Python begins ordering its rows and columns with 0.*

To test your code let `E1` be the the edge list E above (formatted as a 10×2 NumPy array). Then the output of the function call `adj_matrix(5,E1)` should be the matrix A above (formatted as a 5×5 NumPy array). We have included `E1` in the notebook for your convenience

Draw a few more simple networks, and compute their adjacency matrices by hand. Compare the answers you get with the output of your function to further confirm that it's working.

4. Degree Centrality

Recall, one naive way to measure the "importance" of a website is by counting how many other websites link to it. Consider the network G' in Figure 7, which we interpret as consisting of the search results from a websearch.

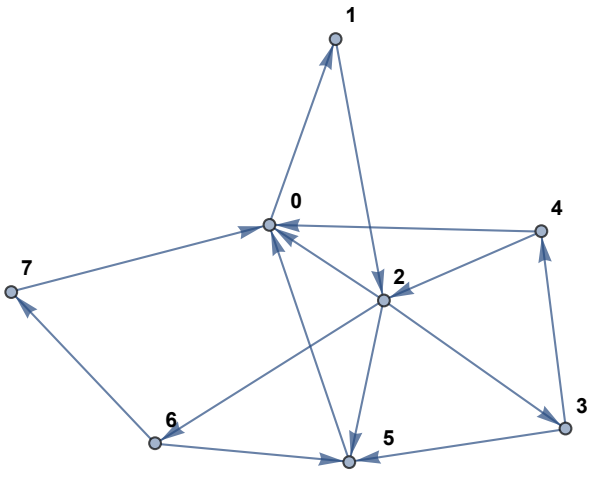

Figure 7. A directed network G' of search results.

The edge list used to describe this network is

$$E' = \begin{bmatrix} 0 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 4 & 4 & 5 & 6 & 6 & 7 \\ 1 & 2 & 0 & 3 & 5 & 6 & 5 & 4 & 2 & 0 & 0 & 5 & 7 & 0 \end{bmatrix}^T$$

(here we are displaying the transpose of the 2×14 edge matrix to save space).

The adjacency matrix of this network is

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

Suppose we are numbering our rows and columns of the matrix A' from 0 to 7. Then recall that the entry in the ith row and jth column represents the number of links from webpage i to j. Thus, if we want to know how many webpages link *to* webpage j, it suffices to add up all of the entries in column j of the adjacency matrix A'. Thus we can rank our webpages based on the sum of the entries in the corresponding column of A'. We call the resulting sum of entries the *in-degree* of the corresponding vertex (i.e. the number of edges that come into the vertex), while the number of edges leaving a vertex is called the *out-degree*. This measure of importance of a vertex is called *degree centrality*.

Problem 2. Define a function called `degree_cent(n,edge_list)` which accepts as input a number of vertices `n` and a list of edges `edge_list` (which is formatted as a $m \times 2$ NumPy array as in Problem 1) and returns a 1-dimensional NumPy array `deg_array`, whose ith entry is the in-degree of the ith vertex of the graph defined by the list `edge_list`. Again you may assume the vertices of the graph referred to in `edge_list` are labelled by the integers `0`, `1`, $\cdots$, `n-1`.

Hint: Recall that you can access the jth column of an array `A` *by* `A[:,j]`. *Also, note that the command* `np.sum(X)` *will return the sum of the values in the array* `X`.

For example, if `E1` is a NumPy array representing the edge list E (from the network G before Problem 1) then the function call `degree_cent(5,E1)` should return the array

```
array([1., 1., 3., 1., 4.])
```

While if `E2` is the edge list E' from the network G' above then making the function call `degree_cent(8,E2)` should return the array

```
array([4., 1., 2., 1., 1., 3., 1., 1.])
```

Judging from the results of the function `degree_cent` above, we can see, using the measure of degree centrality, that vertex 4 is the most significant vertex in network G, while vertex 0 is the most significant vertex in network G' (both of them have in-degree 4, which is larger than the in-degree of any other vertices). Similarly vertices 0, 1, and 3 are the most insignificant vertices of network G, while 1, 3, 4, 6, and 7 are the least significant vertices in network G'.

We can already see how the measure of degree centrality fails to differentiate the importance of a large portion of the vertices.

We would like to look at a network and determine the most important vertex by degree centrality. Determining the relative significance of the vertices in G and G' was easy, because they only contained 5 and 8 vertices respectively, so the list we had to look through only had 5 and 8 entries respectively. To select the most important vertices when the number of vertices is much higher will be more difficult (the network we care about has 499 vertices).

Instead of inspecting the entire list which is returned by `degree_cent`, we can use the function `.argsort()`. To see how `.argsort()` works, consider the following code

```
my_array=np.array([1,3,2,5,0])
my_array.argsort()
```

```
array([4, 0, 2, 1, 3])
```

Notice that unlike other functions we've seen before, we call the function `.argsort()` by appending it to the name of the array that we are applying it to.

The output of the command `my_array.argsort()` is an array whose first entry tells us the index of the smallest entry in `my_array`, while the second entry tells us the position of the second smallest entry of `my_array`, and so on. In other words, the smallest entry in the array `my_array` is in position 4 (which is 0), the second smallest is in position 0 (which is 1), and so on. Reading from the end of the list, we see that the largest entry in the array `my_array` is in position 3, the second largest is in position 1, etc.

We can therefore use the function `.argsort()` to find the location of the larges entries in a long 1-dimensional NumPy array. Instead of looking all the way to the end of the output array, we can reverse the order of `my_array.argsort()` using the function `np.flip()` which reverses the order of an array. Combining these two functions gives us

```
my_array=np.array([1,3,2,5,0])
np.flip(my_array.argsort(),0)  # The 0 here is a required argument.
```

```
array([3, 1, 2, 0, 4])
```

---

which tells us that the largest entry in the array `my_array` has index 3, the second largest has index 1, and so on.

Problem 3. Use the function `degree_cent()`, along with the functions `.argsort()` and `np.flip()` to determine the index of the website with the highest in-degree for the data contained in the array `webpage_data`. Save the number corresponding to the website with the highest in-degree as the variable `top_indegree`.

Hint: Don't forget to include the `0` *in the function call for* `np.flip`.

For example, if you were answering the same question for the graphs G and G', the values you would save in the variable `top_indegree` would be `4` and `0` respectively.

Note that instead of using the command `np.flip` in the above function we could access the last entry using the index `-1`. However, if there are more than one vertices which are tied for the largest value this will only return one of them. This is why we use the function `np.flip` above.

5. PageRank Centrality

Recall that the notion of degree centrality doesn't seem to be a very refined measure of importance, as it failed to distinguish the vertices 1, 3, 4, 6, and 7 in the graph G'. Here we will introduce a refinement of this notion, which will allow us to distinguish these vertices by their relative importance. To introduce this idea, we will use the network G' from above. We'll denote the vertices of G' by $v_0, v_1, \ldots, v_7$.

Suppose the importance of vertex i is denoted by x_i. So, for example, v_0 has importance x_0, while v_1 has importance x_1, and so on. Now, if there is a link from v_i to v_j, we think of v_i as passing some of its importance to the vertex v_j, by virtue of the fact that v_i links to v_j. However, the amount of importance that v_i passes to v_j depends on the total number of links going *out* of v_i. If v_i has m links going out of it, then it passes off $\frac{1}{m}$th of it's importance to each of the vertices it links to.

To make things more concrete, consider the vertex v_2, which has importance x_2. Because it has four links to other webpages, it passes importance of $\frac{1}{4}x_2$ to each of the vertices v_0, v_3, v_5, and v_6.

On the other hand, v_2 has two links coming into it, from v_1 and v_4, and hence v_2 receives importance from v_1 and v_4. Since v_2 is the *only* webpage v_1 links to, v_1 contributes all of its importance to v_2. Since v_4 links to two different webpages, it only passes on *half* of its importance to v_2. In other words, we have

$$x_2 = x_1 + \frac{1}{2}x_4.$$

We can run through a similar argument for the other vertices. Doing this, we obtain the following system of equations:

$$\begin{aligned}
x_0 &= \frac{1}{4}x_2 + \frac{1}{2}x_4 + x_5 + x_7 \\
x_1 &= x_0 \\
x_2 &= x_1 + \frac{1}{2}x_4 \\
x_3 &= \frac{1}{4}x_2 \\
x_4 &= \frac{1}{2}x_3 \\
x_5 &= \frac{1}{4}x_2 + \frac{1}{2}x_3 + \frac{1}{2}x_6 \\
x_6 &= \frac{1}{4}x_2 \\
x_7 &= \frac{1}{2}x_6
\end{aligned}$$

Notice that we can rewrite the above system as a matrix equation as

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1/4 & 0 & 1/2 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/4 & 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}.$$

Setting

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} 0 & 0 & 1/4 & 0 & 1/2 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/4 & 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}$$

this equation can be written more succinctly as $P\mathbf{x} = \mathbf{x}$. In other words, *the vector* $\mathbf{x}$ *that we are looking for is an eigenvector of the matrix* P *corresponding to the eigenvalue* $\lambda = 1$*!* While we don't know that such an eigenvector yet exists for P, we notice that all of the entries of P are nonnegative, and that the columns of P all add up to 1 (similar to the matrices we saw when studying Markov chains in Lab 9). We call a matrix with all nonnegative entries, whose columns sum to 1 a *stochastic matrix*. An important fact about stochastic matrices is that their dominant (largest) eigenvalue is always equal to $\lambda = 1$.

Returning to our website ranking problem, this fact implies that we can indeed find a vector $\mathbf{x}$ which satisfies $P\mathbf{x} = \mathbf{x}$. We call this vector the *PageRank centrality* of the network G', and we can interpret the entries of $\mathbf{x}$ as telling us the importance of the individual vertices (webpages) of the network. Moreover, because $\lambda = 1$ is the dominant eigenvalue, we can find the corresponding eigenvector $\mathbf{x}$ via the iterative methods we studied in Lab 9.

Even better, if we want to find $\mathbf{x}$ by iterative methods, we can start with an initial guess $\mathbf{x}_0$ whose entries all add up to 1. For example, we can set our initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 1/8 \\ 1/8 \\ 1/8 \\ 1/8 \\ 1/8 \\ 1/8 \\ 1/8 \\ 1/8 \end{bmatrix}.$$

Because the columns of P all add up to 1, the vector we get by taking the product $P\mathbf{x}_0$ will also have entries that add up to 1. Likewise with $P^2\mathbf{x}_0$, and $P^3\mathbf{x}_0$, and so on. So when performing our iterative approximations *we don't need to normalize* like we did in Lab 9.

Taking a large power of our starting approximation, say $P^{100}\mathbf{x}_0$, gives

$$P^{100}\mathbf{x}_0 = \begin{bmatrix} 0.227273 \\ 0.227273 \\ 0.242424 \\ 0.0606061 \\ 0.030303 \\ 0.121212 \\ 0.0606061 \\ 0.030303 \end{bmatrix}.$$

Comparing this to $P^{101}\mathbf{x}_0$ gives us very close values (within 10 decimal places) so we may conclude that the iterations converge, and that our PageRank centrality vector is given by $\mathbf{x} \approx P^{100}\mathbf{x}_0$. Notice that if we interpret these values as indicating the relative importance of the websites, they provide a much more refined measure than we were able to get by considering the degree centrality alone.

The final observation to make is about the matrix P, and how it was obtained. Comparing it to the adjacency matrix of our network G', which was

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

we see that P was obtained by taking the transpose of A', and then dividing each column by the sum of the entries in that column.

Problem 4. Define a function called `stoch_mat(A)` which takes as input a square NumPy array `A`, takes the transpose of it, and then divides each column by the sum of the entries in that column. The output of this function will be the resulting NumPy array. You may assume that none of the rows of `A` have negative entries, or sum to zero.

If `A2` is the matrix A' above (formatted as an 8×8 NumPy array), then the function call `stoch_mat(A2)` should return

```
array([[0. , 0. , 0.25, 0. , 0.5 , 1. , 0. , 1. ],
       [1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0.5 , 0. , 0. , 0. ],
       [0. , 0. , 0.25, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0.5 , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0.25, 0.5 , 0. , 0. , 0.5 , 0. ],
       [0. , 0. , 0.25, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0.5 , 0. ]])
```

Hint: You can transpose a NumPy array `A` *by* `np.transpose(A)`.

Problem 5. Define a function `stoch_eig(P,k)` which takes as input an $n \times n$ stochastic matrix `P` (formatted as a NumPy array) and an integer `k`, and returns an approximation for the dominant eigenvector of `P` after `k` iterations. Your starting approximation for the dominant eigenvector should be a NumPy vector `x_0 = [1/n,1/n, ... , 1/n]` with `n` entries.

Hint: Since `P` *is a stochastic matrix and your starting approximation has entries which sum to* `1`*, you* do not need to normalize *after each iteration. In other words, the output of your function using* `k` *iterations will simply be* $P^k x_0$.

For example, if `P` is the matrix P above, then the function call `stoch_eig(P,100)` should return the array

```
array([0.22727273, 0.22727273, 0.24242424, 0.06060606, 0.03030303,
       0.12121212, 0.06060606, 0.03030303])
```

Problem 6. Define a function called `PageRank_cent(n,edge_list,k)` which accepts as input a number of vertices `n`, a list of edges `edge_list` (which is formatted as a 2-dimensional NumPy array as in Problem 1), and an iteration number `k`, and performs the following:

(1) Creates the adjacency matrix `A` of the network with `n` vertices and edges in `edge_list`,
(2) Creates the stochastic matrix `P` corresponding to the adjacency matrix `A` (you may assume here that none of rows of `A` sum to zero),
(3) Approximates the dominant eigenvector of `P` after `k` iterations.

Your function should return the approximation of the dominant eigenvector after `k` iterations (formatted as a 1-dimensional NumPy array).

Hint: As always, your function may call the functions you wrote in the previous problems.

For example, the function call `PageRank_cent(8,E2,100)` (where `E2` is the NumPy array corresponding to the edge list E' above) should return the array

```
array([0.22727273, 0.22727273, 0.24242424, 0.06060606, 0.03030303,
       0.12121212, 0.06060606, 0.03030303])
```

Problem 7. Use the function `PageRank_cent` to find the PageRank centrality vector for the data in `webpage_data` using 100 iterations. Save the index of the webpage with the highest PageRank centrality as the variable `top_PageRank`.

Hint: You may again use the commands `.argsort()` *and* `np.flip()` *to find the location of the largest entry in the eigenvector.*

For example, if you were answering the same question for the network G', the value you would save in the variable `top_PageRank` would be 2.

6. Google, PageRank, and Markov Chains

Note that we can instead interpret the PageRank algorithm above as a Markov chain, similar to the ones we considered in Lab 9. Instead of thinking of x_i as denoting the importance of a webpage, we could instead think of it as the probability that a websurfer will end up at webpage v_i if they randomly click on links to travel from one webpage to another. For example, recall that the only two links in the network G' that land on v_2 are links from v_1 and v_4. Thus the probability of landing on v_2 is comprised of the probability x_1 of landing on v_1, times the probability of selecting the link on v_1 which directs to v_2 (that probability is 1, since there is only one link on page v_1) plus the probability x_4 of landing on v_4, times the probability of selecting the link on v_4 which directs to v_2 (which is $\frac{1}{2}$ since there are two outgoing links on v_4). In other words

$$x_2 = x_1 + \frac{1}{2}x_4$$

which is the same equation we obtain above using our alternate interpretation. We can thus think of the PageRank vector as being a vector whose ith entry indicates the long term probability that a websurfer will be on webpage v_i at some given time.

In the above lab we have described a simplified version of the PageRank algorithm. In practice, Google uses a slightly modified algorithm, which accounts for *dangling nodes*, which are webpages which have no outgoing links. A websurfer who randomly arrives at such a website would never leave (assuming they are only able to visit other webpages by clicking links on their current webpage). To account for these situations, the full PageRank algorithm includes a *damping factor*, which, roughly speaking, adds in the possibility that a websurfer will randomly pick another webpage altogether (instead of clicking a link on the current page).

The PageRank algorithm is known to converge quite quickly. In their original paper, Brin and Page reported that on a network with 322 million edges the algorithm converged to usable values within 52 iterations.

Finally, as a historical note, the patent for the PageRank algorithm is owned by Stanford University (where Brin and Page were students at the time they developed it). Stanford granted Google exclusive license rights to use the algorithm, in exchange for 1.8 million shares of Google which Stanford sold in 2005 for $336 million. Today those shares would be worth approximately $3.8 billion. All for an algorithm which computes an eigenvector!