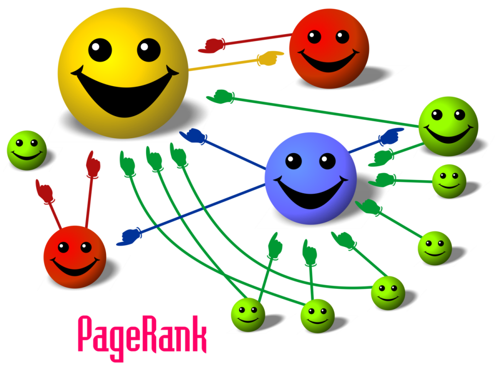
The Page Rank Algorithm
ML_2022: Machine Learning

https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/pagerank_lecture/pagerank_lecture.pdf



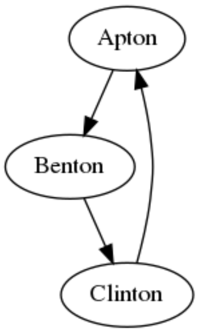
The size of each node reflects its importance, and is related to the importance of the nodes pointing to it. [Image from Wikipedia]

The Page Rank Problem

Web browsers accept a search item from the user, and return within a few seconds with a list of pages that seem to correspond to the search item. This is impossible.

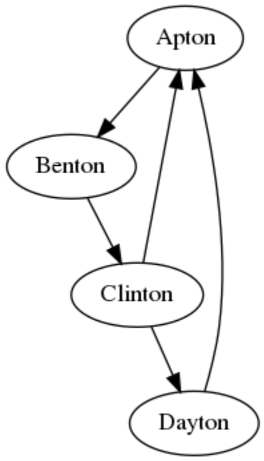
- *There are about 2 trillion web pages on the Internet;*
- *A user expects the search engine to return the web pages which are the closest match to their item, and which are of the highest quality;*
- *The search engine cannot judge quality by "reading" the pages;*
- *The search engine cannot expect human assistance in identifying the best matches.*

1 Feeding the cows



A simple three-field system for grazing cows.

A farmer's cows have three fields to graze in: *Apton*, *Benton*, and *Clinton*. The cows stay in each field for a day. There are one-way gates that connect Apton to Benton to Clinton to Apton, and each morning the cows move to the next field, where the farmer has already delivered the day's supply of food. There are 100 cows, and each cow eats 10 pounds of special feed a day, so the farmer's schedule is a simple rotation of delivering 1000 pounds of feed to Apton, then to Benton, then to Clinton, and back to Apton.



The blasted four-field system for grazing cows.

The farmer is able to buy an extra field called *Dayton.* There is a gate leading from Clinton to Dayton, and another gate goes from Dayton to Apton. When the cows are leaving Clinton, they now have two choices: go to Dayton or Apton. Typically, half go one way and half the other. The farmer figures that on the fourth day, instead of dropping 1000 pounds of feed at Apton, he will drop 500 at Dayton and 500 at Apton, because that's how the cows split up. On day five, he will have to drop 500 at Apton and 500 at Benton, on day 6, the same at Benton and Clinton. On day 7, half of the cows in Clinton go to Dayton and half to Apton, and this is just getting too complicated.

The farmer is sorry he bought the field. Can we help calculate what is going to happen, and whether his life is going to continue to get more and more complicated? We start filling in a table, and the farmer is **not happy** to see what we predict will happen on day 10!

Day	Apton	Benton	Clinton	Dayton	
1	100	0	0	0	
2	0	100	0	0	
3	0	0	100	0	
4	50	0	0	50	
5	50	50	0	0	
6	0	50	50	0	
7	25	0	50	25	
8	50	25	0	25	
9	25	50	25	0	
10	12.5	25	50	12.5	cowmageddon?

Of course, 12.5 cows is just a mathematical model for what is going to happen; we're not going to see any cow split in half. The interesting thing is that in this new system, the cows seem to be scattering among the fields. Is this a random sort of scattering, or is there a pattern?

2 Using graph theory

A *graph* is a set of nodes, labeled 1 through n, and edges, which are pairs of node labels. In a graph, the edge (i, j) means there is a link between nodes i and j which can be traveled in either direction. The adjacency matrix A for such a graph has a 1 in both entries $A_{i,j}$ and $A_{j,i}$ for each edge (i, j).

A *directed graph*, also called a "digraph", is, as before, a set of nodes and directed edges. The directed edge (i, j) means there is a link from node i to node j, which can only be traveled in that direction. If travel is possible in either direction, we must list both (i, j) and (j, i) as directed edges of the directed graph. The incidence matrix A for a digraph has a 1 in entry $A_{i,j}$ corresponding to a directed edge (i, j).

Clearly, to discuss the farmer's problem, we are dealing with a digraph. Here is the incidence matrix for the four field grazing area:

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

We will also be interested in something called the *transition matrix*. Given an incidence matrix A, the transition matrix T is found by

1. copy $T \leftarrow A$;
2. if row i is completely zero, set $T_{i,i} = 1$;
3. dividing each row by the sum of its entries;
4. transposing the resulting matrix.

For our four-field problem, we have:

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

If we have determined the adjacency matrix A, we can automate the computation of the corresponding transition matrix T using Python:

```
def transition_from_adjacency ( A ):
  import numpy as np
  T = A.copy()      # T = A doesn't do what we would expect!
  m = T.shape(0)
  for i in range ( 0, m ):
    if ( np.sum ( T(i,:) == 0 ):
      T(i,:) = 1
    T(i,:) = T(i,:) / np.sum ( T(i,:) )
  T = np.transpose ( T )
  return T
```

The transition matrix has several properties we will need later:

1. it is square;
2. all entries are nonnegative;
3. each column sums to 1

A matrix with these properties is often called a *stochastic matrix* or a *Markov matrix*, because it has important uses in probability, statistics, and simulation.

Our transition matrix has two interesting uses:

- it can be used to model a random walk of one person along the graph;
- it can be used to model the "flow" of many people along the graph;

In particular, suppose that the vector x_1 is a list of the number of cows at each location on day 1. As it turns out, two cows died, so we are down to 98 cows. Set $x_2 = T * x_1$, and $x_3 = T * x_2$ and so on. We don't have to work this data out; instead, we can write a Python script *cows.py* to do it for us:

```
def cows ( ):
  import numpy as np
  T = np.array ( [ \
    [ 0, 0, 0.5, 1 ], \
    [ 1, 0, 0,   0 ], \
    [ 0, 1, 0,   0 ], \
    [ 0, 0, 0.5, 0 ] ] )
  for it in range ( 0, 61 ):
    if ( it == 0 ):
      x = np.array ( [ 98, 0, 0, 0 ] )
    else:
      x = np.dot ( T, x )
    print ( x )
  return
```

```
 1  98.0000         0         0         0
 2        0   98.0000         0         0
 3         0         0   98.0000         0
 4  49.0000         0         0   49.0000
 5  49.0000   49.0000         0         0
 6        0   49.0000   49.0000         0
 7  24.5000         0   49.0000   24.5000
 8  49.0000   24.5000         0   24.5000
 9  24.5000   49.0000   24.5000         0
10  12.2500   24.5000   49.0000   12.2500
...  ...       ...       ...       ...
20  24.5000   32.1562   30.6250   10.7188
30  28.3281   29.2852   26.9883   13.3984
40  28.3401   27.9932   27.5625   14.1042
50  28.0560   27.9019   27.9722   14.0699
60  27.9877   27.9773   28.0262   14.0087
```

It turns out this process is converging to the solution $x = [28, 28, 28, 14]$. Does this make sense? Looking at the arrangement of fields, this works perfectly. Although the cows move everyday, the number of cows in each field stays the same. This is a **steady state** solution, and is something like a river that flows back around into itself.

Thus we can tell the farmer that, in about two months, the feeding pattern will have settled down, and he can deliver 280 pounds of feed to Apton, Benton, and Clinton fields every day, and 140 pounds to Dayton.

The process of repeatedly multiplying a vector x times a matrix T is an example of an important algorithm from linear algebra, known as the **power method**. The idea is that we are seeking a solution vector x to the eigenvalue problem: $A * x = \lambda x$. For our problem, we have $A = T$, a transition matrix, and we can also assume that $\lambda = 1$. To seek x, we start with a random vector x^0, and repeatedly compute $x^{i+1} = T * x^i$. In most cases, we also normalize the vector at each step, because otherwise it tends to blow up or shrink to zero. However, for our situation, using a transition matrix, we are guaranteed that we don't need to do this. Another problem can arise if the matrix has complex eigenvalues; again, for the transition matrix T, this will not happen. Instead, when we carry out the power method iteration for a problem involving a transition matrix, we can expect that the sequence of vectors x^i will eventually converge smoothly to the exact solution x.

```
import numpy as np
for it in range ( 0, it_max ):
  if ( it == 0 ):
    x = np.random.rand(n)
  else:
    xold = x.copy()
    x = np.dot ( A, xold )
  lambda = np.linalg.norm ( x )
  x = x / lambda
```

Listing 1: The power method for a general matrix A.

But now let's go back to the Apton-Benton-Clinton field and try the same power method. Instead of converging, we saw that the vector $x = [100; 0; 0]$ produces a repeating cycle of distinct values. The power method doesn't give us a steady solution for this starting value. We know that there is one: just start with $\frac{1}{3}$ of the cows in each field. So for that problem, using the "wrong" starting value will mean that we never get the answer we are looking for.

This suggests that we can't be sure that the power method, given a random starting vector, will converge to a steady flow on our network. Can someone please tell me if we can fix this?

3 Here comes the math!

The Perron-Frobenius Theorem (version 1):
If A is a square matrix whose entries are all strictly positive, then the largest eigenvalue is real, positive, and simple (not repeated). All other eigenvalues are strictly smaller in norm. There is a corresponding eigenvector whose entries are all strictly positive.

The Perron-Frobenius Theorem (version 2):
If A is a square matrix whose entries are all strictly positive, and each of whose columns sums to 1, then the largest eigenvalue is 1. This eigenvalue is simple. All other eigenvalues are strictly smaller in norm. There is a corresponding eigenvector whose entries are all strictly positive.

The Perron-Frobenius Theorem (version 3):
If A is a square matrix whose entries are all nonnegative, and the matrix is irreducible, and each of the columns sums to 1, then there is an eigenvalue of 1, and there is no larger eigenvalue. This eigenvalue is simple. There is a corresponding eigenvector whose entries are all nonnegative.

The first two versions of the theorem don't quite apply to our transition matrix T, because it includes zero entries. The third version will apply if our matrix is irreducible. The word *irreducible* looks scary here, but for our purposes, it just means that our graph must be **strongly connected**, that is, you must be able to start at any node and reach any other node. This is certainly true for the two examples we have discussed.

Because version 3 of the Perron-Frobenius theorem applies to our transition matrix T, there must be a eigenvector x corresponding to the eigenvalue 1, so that $x = T * x$. And indeed, that's what we found. This means that we could figure out a permanent feeding routine for **any** configuration of fields, as long as the fields are strongly connected.

However, since we are stuck with version 3 of the theorem, we don't know that 1 is the only eigenvalue of magnitude 1. If there are other eigenvalues of the same size, then the power method will generally not converge to the desired solution. Could this explain our issues with the three field problem?

Let's ask Python:

```
from abc_incidence import abc_incidence
from transition_from_incidence import transition_from_incidence
import numpy as np
```

```
A = abc_incidence()
#
#  Compute the transition matrix.
#
T = transition_from_incidence ( A )
#
#  Request the eigenvalues of T.
#
val = np.linalg.eigvals ( T )
print ( val )
#
#  Compute the norms of the eigenvalues.
#
val_abs = np.abs ( val )
print ( val_abs )
```

Listing 2: abc_check.py checks eigenvalues of the three-field graph.

We get the following results:

```
Val =

  -0.5000 + 0.8660i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i  -0.5000 - 0.8660i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i

val_abs =

    1.0000
    1.0000
    1.0000
```

Our three-field matrix has three eigenvalues of norm 1. Two of them are imaginary. If our starting vector includes components of the corresponding eigenvectors, they will never disappear. The power method cannot converge to the eigenvector associated with the eigenvalue of 1, because of the presence of the two imaginary eigenvalues of equal norm.

In practice, imaginary eigenvalues are associated with regular oscillations in the solution. That's exactly what we see in our three-field problem.

This means that version 3 of the Perron Frobenius theorem is not enough to guarantee that the power method will give us what we are looking for if we want to analyze a graph.

4 The Search Problem

It is hard to explain how impossible the Internet search problem is. When you search for ”banana cake”, a search engine

1. guesses that you are most likely interested in food or recipes, and not the movie “Bananas”, or the rock band “Cake” or the expression “going bananas”;
2. out of all the web pages in the universe, identifies those that seem to match your interest.
3. offers to show you any of these pages, starting with the “most important” ones.
4. retrieves and displays any page you select;

Today, we will only talk about part of item #3:

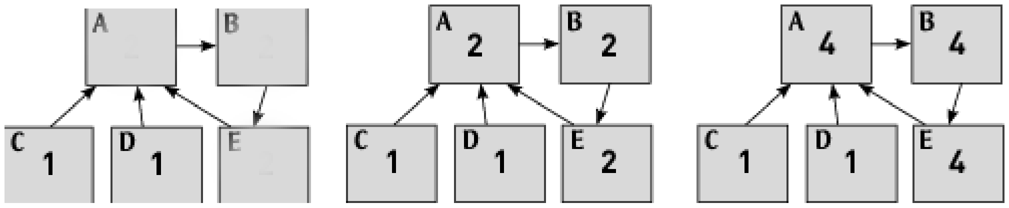
Once the search engine has a list of all the hundreds of thousands of web pages that are relevant to your search, how does it know which pages to show first?

While we think about this question, we must realize that Google can search pages in any language. In step #2, Google does look at matching words in the pages, but the ranking operation does not involve the page content at all! At that point, the pages could be written in a Martian language.

The idea seems simple at first. Web pages can link to other web pages. Therefore, we can try to assign importance based on these links.

The simplest idea is that the rank or importance of a web page is equal to the number of links pointing to it. But surely not all links are equal. If one web page is pointed to by 10 "unimportant" (low ranked) web pages, and the other is pointed to by 10 important web pages, we would probably think the second page is more important, because it collects the importance of the pages that point to it.

So a better idea would be to assign importance by transferring it to a web page from all the web pages linking to it. To get started, we might give an initial important of 1 to any web pages that has no inlinks, and then work from there. But in most cases, this recalculation becomes an endless task.



Updating importance estimates for the ABCDE network becomes an endless chase.

Instead of chasing these numbers around the directed graph forever, let's realize that what we want is a vector of importance values x so that they "flow" around the network in an unchanging pattern. That is, we are, once again, looking for a solution to $x = T * x$, where T is the transition matrix for the network. This is a problem we know how to solve. Python offers the `eig(A)`????? command, or we could simply try to use the power method, repeatedly executing $x \leftarrow T * x$.

This approach will work fine with textbook examples. However, there are still some simple problems that we need to deal with before we can confidently handle a general network.

5 The Google modification

According to the simple ranking rule we have imposed, the importance values of the ABCDE network are `x=[1;1;0;0;1]`, that is, pages C and D have no importance, because no one links to them; while A seems more important than B and C, its two extra inlinks are from pages of 0 importance, so we rank it the same as B and C. Now we don't want any page to have zero importance, because there must be some bit of value there, and someone could always randomly choose that page. And A ought to get some credit for the extra inlinks! These minor problems will be easy to fix.

Another problem arises if a page has no outlinks. In the cow example, this would mean that the cows would be stuck in a particular field with nowhere to go next. In our web page model, a user would be viewing a given page, and have no new page to move to. Such a situation will cause problems for our importance rankings.

Another problem that could arise is that, for the Perron-Frobenius theorem to apply, we needed the network to be strongly connected: from any web page, you should eventually be able to reach any other web page by following some path of links. That is not true for our ABCDE example. And we can expect that on the Internet itself, there are many completely separate "islands" of web pages with no connections whatsoever.

We have already mentioned that, when forming the transition matrix, if we come across a row i in the adjacency matrix which is entirely zero, (no outlinks), we can "patch" the transition matrix with an entry $T_{i,i} = 1$. However, the fact that the transition matrix still includes many zero values means that we can only invoke version 3 of the Perron Frobenius theorem; there may be multiple eigenvalues of magnitude 1, and so the power method might not converge to a single eigenvector.

To take care of web pages with no inlinks or outlinks, and disconnected groups of web pages, the Google approach made a simple modification to the transition matrix model. We imagine a user is currently viewing a given web page. Most of the time, the user's next move is to move to a page pointed to by the current page (assuming there is at least one.) But sometimes, the user just jumps to a randomly selected web page.

In our new model, suppose the user is on a given web page. Then, with probability p, the next thing the user will do is jump to a random web page. With probability $1 - p$, the user will randomly choose a link on the current page and move there. Repeat this many times. The importance or ranking of a page is then the number of times the user visited it.

One is simply to realize that the jumping process to a random link can be modeled by changing our incidence matrix. Starting with the classic matrix A, let let $\mathbf{1}$ represent an $n \times n$ matrix of 1's. If our probability of jumping to a random page is p, then we can modify our incidence matrix to

$$A_p \leftarrow A + p * \mathbf{1}$$

Now every entry of A_p is nonzero, but some are $1 + \frac{1}{p}$ and some are $\frac{1}{p}$. We can interpret these values as the width or capacity or probability of using each link. Let T be the transition matrix of our original matrix A. Then the "Google transition matrix" will be:

$$G = (1 - p) * T + \frac{p}{n} * \mathbf{1}$$

Notice that the effect of adding random jumping is to make every entry of the matrix G strictly positive. This means we can rely on version 2 of the Perron Frobenius theorem: there is a simple eigenvalue equal to 1, all other eigenvalues are strictly smaller, and so the power method will converge and we will get a unique eigenvector solution no matter what starting vector we use.

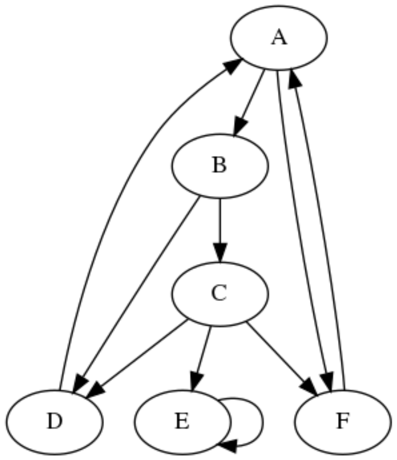
Once we have the matrix G, then we can use the `eigvals()` function or the power method to seek the eigenvector corresponding to the value 1.

Another approach is to simply simulate the user's journey of jumps and moves. This way is called the "web surfer" model. Because the web is enormous, it is not practical to actually try to form the incidence and transition matrices. This is one example of what is now called "big data", that is, problems that are so large that simply storing or representing all the information becomes a major task by itself. Rather than trying to work on a representation of the web, we just use the web itself. We assume we can find any random page, and that on any page we can choose a random link. In that case, we have the "web surfer" ranking approach:

```
def surf_rank ( A )         # Estimate page ranks
  n = A.shape[0]            # Set number of web pages
  p <-- 0.15                # Set jump probability
  r <-- n vector of 0       # Set all page counts to 0
  i <-- index fo random page
  for 0 <= j <= steps
    if j == 0 or rand() < p
      i <-- any random page in network
    else
      i <-- random link j referenced by current page
    r(i) <-- r(i) + 1
  r <-- r / steps
  return r
```

6 Example: A tiny network

As an example of the ranking calculations, consider this 5-node system studied by Cleve Moler:



A 5-node (not simply connected!) network.

Using the transition matrix T, we take 100 power method steps, and then look at the next iterates:

```
          r100          r101          r102

   1:     0.011         0.011         0.010
   2:     0.005         0.005         0.005
   3:     0.003         0.003         0.003
   4:     0.004         0.004         0.004
   5:     0.970         0.971         0.972
   6:     0.007         0.006         0.006
```

The values haven't settled down. The ranking for node #5 is increasing, and the others are dropping. Imagine the cows again. They can wander randomly from field to field, but if they reach the fifth node, they can never leave. So eventually, that's where all of them will be.

Now let's try the same procedure, but using the Google matrix, which allows for occasional random jumps:

```
          r100          r101          r102

   1:     0.235         0.235         0.235
   2:     0.124         0.124         0.124
   3:     0.078         0.078         0.078
   4:     0.100         0.100         0.100
   5:     0.314         0.314         0.314
   6:     0.147         0.147         0.147
```

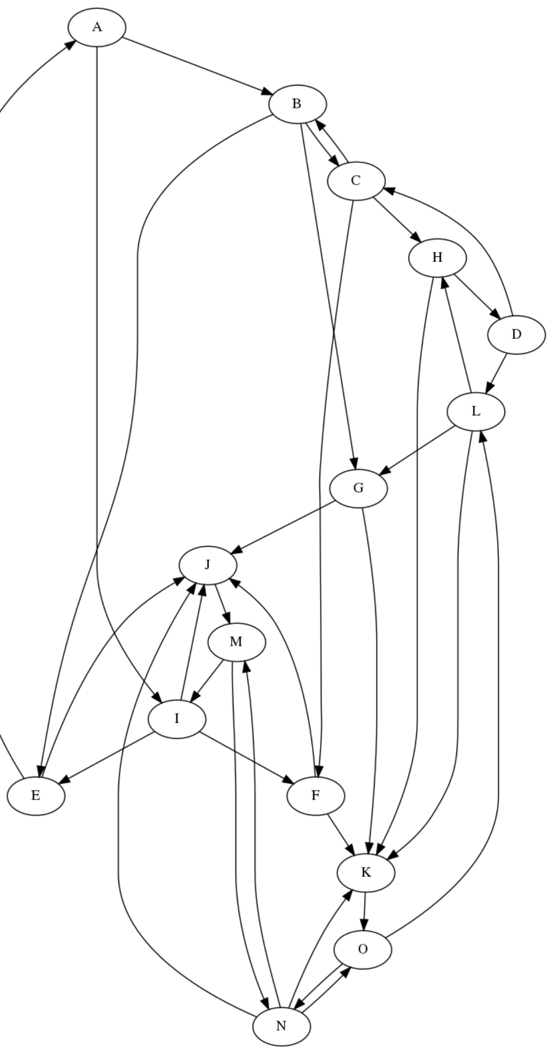
The first thing to note is that the iteration has converged. Node #5 no longer traps all the cows or all the users. The random jumps allow them to leak away to other nodes. Secondly, node #5 still has very high rank, but node #1 has comparable ranking, based roughly on the sum of the rankings of nodes #4 and #6. While having no outlinks seems to favor node #5, it is no longer the black hole that it represented before. Now we have a more uniform sampling of the network, and a mathematically-based estimate of the page rankings.

7 Example: Sauer's internet

In the textbook, *Numerical Analysis* by Timothy Sauer, the following adjacency graph is shown:

```
        1     2     3     4     5     6     7     8     9    10    11    12    13    14    15

  1:    0     1     0     0     0     0     0     0     1     0     0     0     0     0     0
  2:    0     0     1     0     1     0     1     0     0     0     0     0     0     0     0
  3:    0     1     0     0     0     1     0     1     0     0     0     0     0     0     0
  4:    0     0     1     0     0     0     0     0     0     0     0     1     0     0     0
  5:    1     0     0     0     0     0     0     0     0     1     0     0     0     0     0
  6:    0     0     0     0     0     0     0     0     0     1     1     0     0     0     0
  7:    0     0     0     0     0     0     0     0     0     1     1     0     0     0     0
  8:    0     0     0     1     0     0     0     0     0     0     1     0     0     0     0
  9:    0     0     0     0     1     1     0     0     0     1     0     0     0     0     0
 10:    0     0     0     0     0     0     0     0     0     0     0     0     1     0     0
 11:    0     0     0     0     0     0     0     0     0     0     0     0     0     0     1
 12:    0     0     0     0     0     0     1     1     0     0     1     0     0     0     0
 13:    0     0     0     0     0     0     0     0     1     0     0     0     0     1     0
 14:    0     0     0     0     0     0     0     0     0     1     1     0     1     0     1
 15:    0     0     0     0     0     0     0     0     0     0     0     1     0     1     0
```



A plot of Sauer's network.

Now we can apply several methods to assign a rank to each of the nodes. From the incidence matrix A, we can compute the transition matrix T, and the Google matrix G, using a random jump probability of 0.15.

1. We ask Python to compute the eigenvector corresponding to the eigenvalue 1.
2. We can apply the power method.
3. We can try the random surfer technique, starting anywhere, then randomly choosing an existing link, and count how many times we visit each node.

Similarly, we can compute the Google matrix G, using a random jump probability of 0.15. Doing this means we don't have to worry about whether the graph is strongly connected. Again, we can try:

1. Python `eigvals()`
2. Power method.
3. Random surfer, including random jumps with probabiity p;

Sauer's matrix is strongly connected, so if we prefer, we can use the transition matrix. Here are the results using 100 steps of the power method on T, or calling Python's `eigvals()`

```
  #        power         eigvals()

  1:       0.0154      0.0154
  2:       0.0115      0.0115
  3:       0.0115      0.0115
  4:       0.0154      0.0154
  5:       0.0308      0.0308
  6:       0.0308      0.0308
  7:       0.0308      0.0308
  8:       0.0308      0.0308
  9:       0.0810      0.0810
 10:       0.1100      0.1100
 11:       0.1100      0.1100
 12:       0.0810      0.0810
 13:       0.1467      0.1467
 14:       0.1467      0.1467
 15:       0.1467      0.1467
```

Here we see that the power method has a 4-digit match with Python's eigenvalue calculation.

Now let's work with the Google matrix G instead, using the typical probability $p = 0.15$. Using 100 steps of the power method, or Python's `eigvals()` function, or 100,000 steps of the surfer technique, we get the following page rankings:

```
          power     eigvals       surf

  1:      0.0268     0.0268     0.0259
  2:      0.0298     0.0298     0.0303
  3:      0.0298     0.0298     0.0301
  4:      0.0268     0.0268     0.0271
  5:      0.0395     0.0395     0.0392
  6:      0.0395     0.0395     0.0395
  7:      0.0395     0.0395     0.0391
  8:      0.0395     0.0395     0.0403
  9:      0.0745     0.0745     0.0738
 10:      0.1063     0.1063     0.1056
 11:      0.1063     0.1063     0.1070
 12:      0.0745     0.0745     0.0745
 13:      0.1250     0.1250     0.1247
 14:      0.1163     0.1163     0.1164
 15:      0.1250     0.1250     0.1265
```

from which we can conclude that we used enough power method steps, but maybe, if we wanted 4 digit accuracy, the surfer should have used more steps!

Now, with so many choices, which calculation do we prefer? Well, first of all, we need something like the Google modification in order to make the Internet look strongly connected. The power method and direct eigenvalue/eigenvector calculations require having an explicit form for the transition matrix. We can't afford that if we really want to model the Internet. On the other hand, Google can afford to send out multiple web crawlers, which do nothing but surf the Internet. These can be programmed to do the random jump/random link algorithm, and to send back the running totals of the pages they have visited. The results from all the web crawlers can be combined into a single count array, which, when normalized, gives a good estimate of the web page ranking. If we throw out the oldest count information as time passes, then our ranking will automatically keep track with changes on the web.

Thus, the PageRank algorithm allows a search engine company to rank all the web pages on the Internet. And thus, once the search engine has created a list of all the web pages that match your “banana cake” query, it can use the rankings to decide which pages to show first. Thus, the PageRank algorithm is just a part of a much more involved system of quickly and efficiently returning a search result.

These methods are not the only ways of ranking web pages. In the lab exercises, we will also encounter another method, known as the HITS method. In any case, a web browser needs a method of page ranking, so that, once it has found a set of pages which match the user’s search term, it can order them in such a way that the most important or respected or referenced pages show up first.

Because Google was the first browser to solve this problem efficiently (in 1998), it quickly came to dominate the market, and destroyed a number of other much less powerful search engines like Archie, Gopher, JumpStation, Ask Jeeves, AltaVista, Lycos, WebCrawler, Excite, HotBot, and others.