

CSL: PageRank Algorithm

Nicole Beckage and Marina Kogan

May 2, 2013

History of PageRank

We live in an era of informational deluge, some of which is available to us through the World Wide Web. We navigate this abundance of information by querying the web on the key words of interest. For this purpose, we use search engines such Yahoo, Bing, and of course Google. In this lecture we discuss PageRank - one of the algorithms behind Google search engine.

Most of the early search engines used the text based ranking system. They kept an index of all web pages, and when a user typed in a query search, the engine browsed through its index and counted the occurrences of the key words in each web page. The pages displayed at the top of the result list were the pages with the highest number of occurrences of the key words. The problem with this approach is that pages with high occurrence of the key words do not necessarily contain relevant information about the key words. Moreover, suppose we wanted to find some information about the CU. We would type in the word "Colorado" and expect that "www.colorado.edu" would be one of the most relevant sites to our query. However, there may be millions of pages on the web using the word Colorado, and www.colorado.edu may not be the one that uses it most often. For example, if someone decided to write a web site that contains the word "Colorado" a billion times and includes not other information, this web site would be the first one displayed by a search engine. But this is obviously not the most useful result.

The usefulness of a search engine depends on the relevance of the result set it gives back. Some web pages containing the search key words are more relevant, popular, or authoritative than others. So ideally, we would expect the search engine to display the relevant pages at the top of

the result list. One of the most influential algorithms for computing the relevance of a web page is PageRank.

It is called PageRank after Larry Page who came up with the algorithm jointly with Sergey Brin in 1998 ¹when they were graduate students at Stanford University. At the core of PageRank is the notion that the importance of any web page can be judged by looking at the pages that link to it. If we create a web page u and include a link to the web page v , this means that we consider v important and relevant for the topic of our site. If there are a lot of pages that link to v , this means that the common belief is that page v is important. If, on the other hand, v has only one incoming link, but it comes from an authoritative site w , we could say that w transfers its authority to v ; link from w imbues v with importance. Whether we talk about popularity or authority, we can iteratively assign a rank to each web page, based on the ranks of the pages that point to it.

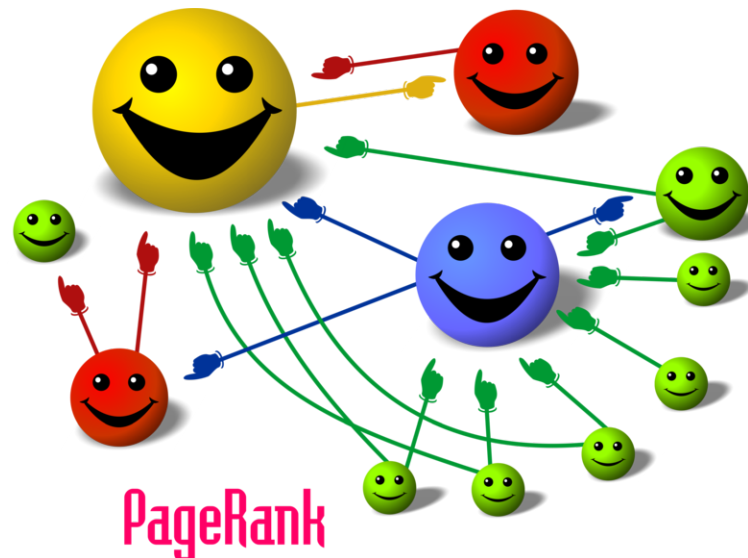


Figure 1: Visual representation of PageRank. Here the size of the nodes represent importance of a node. We can see that the yellow node has the highest importance because it has a lot of incoming links but, importantly, many of these incoming links are from pages that also have other pages of high PageRank pointing towards it. One of the red nodes has only one incoming link but from the yellow node and thus has a high PageRank itself.

Both Page and Brin went on to found Google Inc, which uses the PageRank algorithm (rather

¹Brin, S.; Page, L. (1998). "The anatomy of a large-scale hypertextual Web search engine". Computer Networks and ISDN Systems 30: 107-117. doi:10.1016/S0169-7552(98)00110-X

a close variant thereof) as one of the algorithms for delivering relevant search results on the Google search engine. The process of PageRank has been patented (to Stanford, not Google) and Stanford received 1.8 million shares of Google in exchange for use. These shares were sold in 2005 for \$336 million.

PageRank as a random surfer model (Matrix power method)

If we are interested in the 'importance' or popularity of a page, one simple way of getting at that might be to categorize how often a webpage is visited if we were randomly surfing the web—clicking on hyperlinks blindly and starting on a random new page with some small chance or when we get stuck on a page with no hyperlinks. This equates to a set of infinite random walks on the hyperlink graph G . Here we are (approximating) how the graph would be visited if we let a whole bunch of random surfers click links at random. Suppose for instance, that we have a small Internet consisting of just 4 web sites [www.pageA.com](#), [www.pageB.com](#), [www.pageC.com](#), [www.pageD.com](#), referencing each other in the manner suggested by the graph:

To start we will use the observation that \mathbf{I} is a matrix representation of where we can get if we take no steps this in one step. This has ones along the diagonal and zeros everywhere else. This is the equivalent to the number of ways we could land at each point. Later we will extend this to account for the probability of visiting each page and any initial biases we might place in the set of initial conditions. We can go one step further to consider the graph \mathbf{A} which will tell us what locations we can get to if we take a single step in this graph. Figure 1 shows an example graph \mathbf{A} and below is the matrix representation of \mathbf{A} . Note that for the sake of this paper, we are using the transpose (swapping $A_{i,j}$ for $A_{j,i} \forall i, \forall j$) of the standard graph representation. This is because importance of a page is transferred in the opposite direction of the hyperlink. That is if there is a hyperlink from B to A , A will take on the importance of B so we set the matrix representation $A_{A,B}$ to be 1.

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

We extend this by noting that A^2 gives us the locations (and start/end points) we can get to in two steps as well as the number of ways to get to this location. For intuition, when we look at A^2 we get paths of length 2. You can check the results by referencing the figure above. For example we can see that there are 3 ways to get relevance weight to node C in two steps (path ADC , path BDC and path CDC) and that the row of the matrix below for node C has 3 ones. We can also see that there are three possible paths of length 2 that start from C (path CBA , CDC and CBD) which is why the column has 3 ones. Since none of these paths end at node B we do not

show a path between C and B of length 2. Further we can see that there is no path from D to C that takes exactly 2 steps. We extend this one step further to paths of length 3. Here we can see that there is no path of length 3 between A and itself but a path does exist between B and itself.

$$A^2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \text{ and } A^3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \end{pmatrix}$$

We can repeat this process increasing the power we raise \mathbf{A} to each time and if we sum them all up we, in effect, get the sum of taking an infinite set of walks of varying length.

If we were to do this with a single number we could consider the limit of this series and approximate it through Taylor series expansion. This requires that $|x| < 1$.

$$1 + x + x^2 + x^3 + \dots = \sum_{i=0}^{\infty} x^i = (1 - x)^{-1}$$

which can 'easily' be extended to matrices:

$$\sum_{i=0}^{\infty} G^i = (\mathbf{I} - G)^{-1}$$

where \mathbf{I} is the identity matrix. This is known as the Neumann series. This has a similar restriction that the absolute value of eigenvalues of the matrix G have to be less than 1. We will show that this is (nearly) guaranteed by the construction and modification of \mathbf{A} for PageRank below since the principal eigenvalue is 1 and all other eigenvalues are strictly less in magnitude than the principal (first) eigenvalue.

It should be noted that not all matrices have an inverse and, in fact, it is non-trivial to prove whether an inverse exists (we need to know the determinant of a matrix). Further, even though we have a mathematical solution for a sum of an infinite set of walks on a given graph, computing this solution is not particularly quick. Running the inverse on a matrix of just 1000 nodes takes both matlab and R a few seconds. Thus computing the inverse over a graph representing the entire web is going to take a while and is not going to be a useful algorithm for online results. We also require storing the whole inverse in order to use it to approximate relevance.

Returning to defining a mathematical inverse: By definition invertibility means that given a n -by- n matrix A and its inverse B then

$$AB = BA = \mathbf{I}$$

We only consider square matrices since any input into PageRank will be a square matrix.

As mentioned, the principal eigenvalue (denoted as λ_1) has to be 1 by the construction of G and that all other eigenvalues are less than the absolute value of 1. This allows us to use the Neumann series. Since $\lambda_1 = 1$ and not strictly less than we will have to approximate the inverse as follows:

$$(\mathbf{I} - G)^{-1} \text{ as } (\mathbf{I} - |\text{abs}(\lambda_1) - \epsilon|G)^{-1}$$

which is invertible for any positive $\epsilon > 0$. If we let ϵ approach 0, then the limit becomes $(\mathbf{I} - G)^{-1}$ which has a solution. In practice we take G and multiply it by a value very close to 1 which will allow us to approximate the inverse with enough precision.

Now we know can compute what $(\mathbf{I} - G)^{-1}$ is. But this still does not get us to the PageRank since we want an array (not a matrix) of values back. To do this we want to create a vector that we will call \mathbf{c} . This vector has interpretable meaning in that it allows us to bias where random walks are initially starting (or influence of the untraversed graph). For example if we wanted to say that individuals have to begin navigating from a specific page, we would be able to do this by having vector \mathbf{c} be all zeros except for the starting page which we'd set to 1. In the case of PageRank we don't want to bias where we start from and thus we use a vector with uniform probability across all pages. For easy mapping to the next section, we will let the initialization vector assign each page a $\frac{1-d}{n}$ probability of being learned. The final equation for PageRank is thus:

$$R \propto (\mathbf{I} - G)^{-1} \frac{1-d}{n} \mathbf{1}$$

Where R is the PageRank vector and $\mathbf{1}$ indicates a column vector of 1s. With this specification, the resulting PageRank vector is not normalized and thus not a probability. But the output is proportional to the probability.

1 Probabilistic View of PageRank

As we mentioned before, PageRank is a measure of importance or influence of a web page, in this case estimated by how often or high likely it would be visited by a surfer. This is an inherently iterative measure. Let us see how we would iteratively derive PageRank from our toy WWW example.

Remember that in our example page A links to page D , page B links to A and D , C links to B and D , and, finally, page D links to only C . In our basic model, let us assume that each page transfers its importance evenly to the pages that it links to. Since node A has 1 outgoing edge, so it will pass on all of its importance to D . But node C has two outgoing edges, so it will pass on its importance equally to node D and B . In general, if a node has k outgoing edges, it will pass on $\frac{1}{k}$ of its importance to each of the nodes that it links to. Thus, our example Web could be described by the following system of equations:

$$\begin{cases} R(A) = \frac{1}{2}R(B) \\ R(B) = \frac{1}{2}R(C) \\ R(C) = R(D) \\ R(D) = R(A) + \frac{1}{2}R(B) + \frac{1}{2}R(C) \end{cases}$$

Since PageRank is an iterative measure, the $R(A)$ depends on the $R(B)$ at the previous time step. Thus, we need to add a time-related notation to our equations above.

$$\begin{cases} R_t(A) = \frac{1}{2}R_{t-1}(B) \\ R_t(B) = \frac{1}{2}R_{t-1}(C) \\ R_t(C) = R_{t-1}(D) \\ R_t(D) = R_{t-1}(A) + \frac{1}{2}R_{t-1}(B) + \frac{1}{2}R_{t-1}(C) \end{cases}$$

Let us see what these equations mean in practice by computing the page rank for the first three time steps. Let us assume that initially PageRank is distributed uniformly (equally) among all the pages.

R_0	R_1	R_2	R_3
$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{16}$
$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{4}$
$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{5}{16}$
$\frac{1}{4}$	$\frac{1}{2}$	$\frac{5}{16}$	$\frac{3}{8}$

The results above are consistent with the intuition we might have about our Web graph. Since node D has the most incoming links we would expect it to be the most important node in the graph, which it becomes at time step 3. On the other hand, this important page has only one outgoing link, to page C , which then shares in some of D 's importance. Consistent with this intuitive assumption, node C is the second most important page at the time step 3.

The PageRank system of equations above could also be re-written in terms of probabilities. What would be the probability of getting to page A in one step? Well, we can only get to A from B , and the probability of going from B to A is $\Pr(A|B) = \frac{1}{2}$. The probability of starting at node B initially is $\frac{1}{4}$. Thus, the probability of getting to A at the first time step is:

$$Pr_1(A) = Pr_0(B)Pr(A|B) = 1/4 * 1/2$$

In general,

$$\begin{cases} Pr_t(A) = Pr_{t-1}(B) * Pr(A|B) \\ Pr_t(B) = Pr_{t-1}(C) * Pr(B|C) \\ Pr_t(C) = Pr_{t-1}(D) * Pr(C|D) \\ Pr_t(D) = Pr_{t-1}(A) * Pr(D|A) + Pr_{t-1}(B) * Pr(D|B) + Pr_{t-1}(C) * Pr(D|C) \end{cases}$$

In the first equation of this system, $Pr_t(A)$ is the probability of being at page A at time t , $Pr_{t-1}(B)$ is the probability of being at page B at time $t - 1$, and $Pr(A|B)$ is the probability of transitioning from B to A . Thus, the PageRank at time t depends only on the PageRank at time $t - 1$, and not the whole path that got the surfer to wherever in the graph s/he is at time $t - 1$. This is called a memoryless property. Additionally, the probability of transitioning from one state to another (i.e. $Pr(A|B)$) is the same at any point in time, whether it's the time step 1 or 31. These two properties describe a stochastic process (which means a process based on probabilities) known as Markov Process.

Markov process is described by a sequence of states R_t within the state space S . In our example, $S = \{A, B, C, D\}$ and R_t is where in the Web graph we are at any given time. The evolution of the Markov Process is determined by the probability transition matrix p , where row i and column j :

$$p(i, j) = P((X_1 = i | X_0 = j))$$

The probability transition matrix represents the probabilities of transitioning from any state to any state in the system. The probability transition matrix for our toy World Wide Web is:

	A	B	C	D
A	0	$\frac{1}{2}$	0	0
B	0	0	$\frac{1}{2}$	0
C	0	0	0	1
D	1	$\frac{1}{2}$	$\frac{1}{2}$	0

The entries in this matrix are consistent with coefficients in the PageRank system of equations above. Note that this is a transpose of the way we have normally represented graphs so far is a matrix. This makes sense, since normally we are interesting in recording outgoing links in the adjacency matrix, but since the PageRank is passed on to the node through its incoming connections, here we care for recording the incoming links. Also notice that all the columns of this matrix add up to 1 - this is called column stochasticity. This also makes sense, since the columns represent probabilities of transitioning out of a node. To keep this in the domain of probability, we want the columns add up to 1. So the matrix above is a transposed adjacency matrix, with each column normalized (divided) by its sum.

Now that we have this matrix, we can go from the probabilistic system of equations to a more compact matrix notation. In this notation, we can describe PageRank with the single equation:

$$R_t = pR_{t-1}$$

where R_t is the PageRank vector at time t , R_{t-1} is the PageRank vector at time $t - 1$, and p is the probability transition matrix.

If we iterate through this process enough times, our Markov Process will converge onto a unique stable state, if the process is ergodic and has a finite state space. Ergodic means that all the states are pairwise-reachable, which means there is a path (of any length) from state x to state y , for all x and y . From what we learned about graphs in this course, we know that this would be the case if we had a single strongly connected component. Another aspect of ergodicity is the fact that the process is aperiodic, which simply means that returns to state x from itself occur at irregular times, with no periodicity. If we could establish that our process is ergodic, we would guarantee that iterating through equation $R_t = pR_{t-1}$ yield the true value of the PageRank.

Algorithm

Returns: Probability distribution over all webpages (size n) representing the likelihood that a person would arrive at that page by randomly clicking links.

Takes: A collection of documents (pages, articles etc) and the links between documents as well as an epsilon to check for convergence.

```
PageRank=function(A,d,epsilon) {  
  G=Build_pagerank_matrix(A,d) #below:convert hyperlink graph to good graph  
  R_old=vector(0,length=n)  
  R=vector(1/n, length=n) #initiate PageRank to equal values  
  while (abs(norm(R)-norm(R_old)>epsilon)) { #if R is still changing  
    R_old=R  
    R=G*Rlast #update Pagerank  
    R=R/norm(R) #renormalize page rank  
  }  
}
```

Building the PageRank Matrix

The formation of the matrix that we will use to find the PageRank value plays a crucial role in the success of this algorithm. There are four main features of the matrix used in the algorithm.

1. To begin with $A_{i,j} = 1$ if there exists a link from j to i which in practice is the transpose of what we traditionally think of as a matrix representation of a graph. We define A this way since influence flows in the opposite direction of the hyperlinks.
2. Any nodes that do not have links to any other pages are said to be connected to all pages in the graph, and the connective strength of this is distributed equally across all pages (including itself). Without this modification pages with no out going links would result

in a dead end. If this happens we assume the searcher would pick a new page entirely at random—this is equivalent to assuming a page with no hyperlinks links to everything.

3. Once we have our corrected matrix such that where j links to is in the j th column, we want to normalize each column so that we can say that the PageRank of j is being equally distributed to all of its neighbors. This means that our matrix is now column stochastic, an important feature we will return to later.
4. Finally, we add a bit of noise to the system by adding a specific value to each cell. This is known as the damping factor and is equivalent to adding a finite probability that individuals will stop clicking hyperlinks or surfing the web. This, importantly, causes the matrix to be positive definite (all values are positive) and irreducible. Irreducibility means two things in this case. We cannot divide the matrix into independent disjoint sets and that the resulting graph is strongly connected (a directed path exists between each vertex and all other vertices in the graph)

Pseudocode for building up matrix:

```
Build_pagerank_matrix=function(A, d) {
#takes in hyperlink graph and damping factor
n=length(A[,1]) #number of webpages
An=matrix(0,nr=n,nc=n)
for (i in 1:n) {
  k=sum(A[,i])#sum of column i
  if (sum(k==0)) { #if vertex i is a sink
    An[,i]=1/n #give every page an equal chance of getting visited
  } else { #if it isn't a sink
    An[,i]=A[,i]/k #normalize the outgoing links
  }
}
G=d*An+(1-d)/n #add the probability of stopping search or randomly switching pages
}
```

In matrix notation we let A be the original hyperlink graph and A_t be the normalized hyperlink graph. Define \mathbf{s} to be a vector of length n with 1s indicating sinks.

$$A_t = \frac{A_{ij}}{\sum_k (A_{kj})}$$

$$A_n = A_t + \frac{1}{n} \mathbf{s}$$

$$G = dA_n + \frac{(1-d)}{n} \mathbf{1}$$

In our example this redefinition of \mathbf{A} is shown in steps below:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \\ 1 & \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix} \rightarrow G = \begin{pmatrix} .0375 & .4625 & .0375 & .0375 \\ .0375 & .0375 & .4625 & .0375 \\ .0375 & .0375 & .0375 & .8875 \\ .8875 & .4625 & .4625 & .0375 \end{pmatrix}$$

This transformation of the hyperlink graph to the end graph that we eventually feed to page rank is the most sophisticated part of the algorithm. It is really this transformation that allows us to make any guarantees about the correctness of the algorithm. What we need for the algorithm to be correct is:

1. We need a stationary distribution at the end. That is to say, the values we assign as the PageRank have reached an equilibrium. The system cannot be wildly different between one iteration and the next. If there is a system that changes significantly from one iteration to the next, we will never reach our criterion that the difference between the PageRank at time point t and time point $t + 1$ are close enough that we exit the loop.
2. We need this stationary distribution to have a unique stable state. It's possible that the way in which we initially compute the PageRank (at time 1) could influence what the resulting stable state looks like in the end. That is to say, we could get a different result if we update A first or B first. That means our PageRank values would only be correct some of the time. Thus we do not only need a stable state but also a unique stable state.
3. We need our PageRank to return all positive values that are either a probability distribution over pages or proportional to this distribution. What would a negative value mean in this system?

We've already discussed exactly how the matrices are formed but what does that afford us as far as these requirements are concerned? Luckily, there's a nice mathematical theorem we can reference (but not prove) that allows us to ensure these properties will emerge from the graph as we designed it.

The *Perron-Frobenius Theorem* comes out of linear algebra and was first proved in 1907 and 1912 (respective to author ordering in the theorem name)²³ In the case of our modified hyperlink graph, this theorem guarantees us 1) the principal eigenvalue is 1. 2) the principal eigenvector is finite 3) the principal eigenvector is unique. The idea here is that if we have a matrix that is *column stochastic*, *irreducible* and *ergodic* where ergodic is defined as irreducible and aperiodic (that is the states don't fluctuate between multiple stable distributions, which is shown above)

²Perron, Oskar (1907), "Zur Theorie der Matrices", Mathematische Annalen 64 (2): 248-263, doi:10.1007/BF01449896

³Frobenius, Georg (1912), "Ueber Matrizen aus nicht negativen Elementen", Sitzungsber. Koenigl. Preuss. Akad. Wiss.: 456-477.

then we get some nice properties out. To begin with, column stochastic means that each entry is non-negative and that the sum across a given column equals 1. This is one of the reasons we remove sinks and is the reason that we normalize across columns. We artificially impose this constraint in order to guarantee a correct solution. Now to define irreducible—a graph (and thus a matrix) is said to be irreducible if there exists a directed path between all pairs of vertices. A matrix is irreducible if one cannot write the matrix in the form of:

$$\mathbf{A} = \begin{pmatrix} B & 0 \\ 0 & C \end{pmatrix}$$

Alternatively a Markov chain is irreducible if there is a non-zero probability of transitioning from one state to any other states (does not have to be in just one step). For our graph we guarantee this by adding a probability of stopping the current path and randomly picking a new webpage (this $\frac{1-d}{n} \mathbf{1}$ term that we add to G). We similarly guarantee that the graph is ergodic by creating a finite state space in which the path lengths between a given node and itself share no common divisor greater than 1 for all paths in the graph. Now that we have satisfied the constraints of the Perron-Forbenius theorem we are guaranteed a single stable state vector which is our PageRank vector.

Further the Perron-Forbenius theorem guarantees us that our algorithm will halt and further that it will return the correct solution.

runtime and space analysis

If we utilize G as defined above and as our algorithm builds it then we get a runtime (within the while-loop) of $O(n^2 * n)$ since the matrix is n^2 and the PageRank vector is also size n . We then iterate over this while-loop k times. In practice, when we actually utilize this algorithm for calculating PageRank on the hyperlink graph k is on the order of 10^2 . Nonetheless, we will consider k as a random variable in our runtime analysis. So if we iterate through k times we end up with a total runtime of $O(k * n^3)$ which is not great considering the web graph is on the order of 10^9 .

How might we improve this runtime? One way is to consider the additional probability of selecting a page at random as a constant that we add to each page. In this case we would utilize the graph \mathbf{A}_t and not the graph G . Recalling that \mathbf{A}_t is the normalized and sink-corrected graph which we will assume is represented as an adjacency list and that the constant we are adding to each cell is $\frac{1-d}{n}$ we get that:

$$R_t = \mathbf{A}_t d * R_{t-1} + \frac{1-d}{n} * R_{t-1}$$

which is looped over k times reducing our runtime to $O(k(V + E)n)$. However, in the graph \mathbf{A}_t we have corrected for sinks. When we correct for sinks we increase the density of the graph since every sink adds V edges. So with the graph defined as above, we cannot make any promises on how small $(V+E)$ is since we do not know how many sinks there are and thus cannot say

that the graph is sparse. So instead we might want to utilize the original hyperlink graph (after normalization, \mathbf{A}_n) since we know that graph is sparse—with edges on the order of the number of vertices. Thus our runtime can be further reduced by defining the algorithm as follows (and representing \mathbf{A}_n as an adjacency list)

```
define: array of length n
sinks[j]=1 if j is a sink, zero otherwise
```

$$R_t = \mathbf{A}_n d * R_{t-1} + 1/n * sinks * R_{t-1} + \frac{1-d}{n} * R_{t-1}$$

which gives us a runtime of $O(k(V+E)n)$ where we know that the graph is sparse with $O(n)$ edges.

For space, we have to store the PageRank vector in all cases. In the two conditions with better runtime, we utilize an adjacency list for $O(V+E)$ space (we discussed conclusions we can draw on this representation above. In the shorter runtimes we also need an additional constant of $\frac{1-d}{n}$ and in the final condition another vector to store sink locations. Thus overall we need $O(V+E)$ space.

Relation to Eigenvectors

The above algorithm (as well as the matrix and markov method) all converge to the same set of values. These values happen to be proportional to the principal eigenvector. First we will define eigenvectors and eigenvalues but will not go into mathematical detail. We will then work through the in text examples. Let \mathbf{A} be a square matrix. \mathbf{Q} is an $n \times n$ matrix whose i th column we denote as the eigenvector of \mathbf{A} . Λ is a diagonal matrix with the eigenvalues of \mathbf{A} along the diagonal. Breaking down \mathbf{A} into its eigenvalues and eigenvectors is called eigendecomposition. Below is an example computed using the programming language R as eigendecomposition is not a trivial computation. The values are truncated at two decimal places for ease of displaying.

$$\mathbf{A} = \mathbf{Q} \Lambda \mathbf{Q}^{-1}$$

Recall that

$$\mathbf{A} = \begin{pmatrix} .0375 & .4625 & .0375 & .0375 \\ .0375 & .0375 & .4625 & .0375 \\ .0375 & .0375 & .0375 & .8875 \\ .8875 & .4625 & .4625 & .0375 \end{pmatrix}$$

When we apply eigendecomposition we get

$$\mathbf{Q} = \begin{pmatrix} .21 & .28 & -.43 + .19i & -.43 + .19i \\ .33 & -.44 & -.11 - .52i & -.11 - .52i \\ .63 & .68 & .61 & .61 \\ .66 & -.52 & -.07 + .34i & -.07 + .34i \end{pmatrix} \text{ and } \Lambda = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.66 & 0 & 0 \\ 0 & 0 & -0.01 + .47i & 0 \\ 0 & 0 & 0 & -0.01 + .47i \end{pmatrix}$$

From here we can now make a few observations. As stated earlier, the principal (first) eigenvalue is 1. Second, all eigenvalues have an absolute value less than or equal to 1, satisfying requirements for the use of the Neumann series above. Most importantly the first column of matrix \mathbf{Q} is proportional to the PageRank.

$$\begin{pmatrix} 0.213 \\ 0.338 \\ 0.632 \\ 0.663 \end{pmatrix} \propto \begin{pmatrix} .115 \\ .183 \\ .343 \\ .359 \end{pmatrix}$$

This isn't the first time someone thought of using eigenvectors as a method of measuring centrality in a graph. In fact eigenvector centrality is a common centrality measure in network analysis and has citations (for a very similar variant) from 1953 (Katz centrality).

2 Convergence of interpretations

We've presented 4 different views of PageRank and how to calculate it through these different sets of methods. To summarize the results we recapture them here.

First, the random surfer model returns the correct (approximated) PageRank. We say approximated because as we showed the principal eigenvector of G must be 1 and the Neumann series requires that it is less than one. We fix this by multiplying G by a value extremely close to 1.⁴

$$R \propto (\mathbf{I} - G)^{-1} \frac{1-d}{n} \mathbf{1}$$

$$R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} .0375 & .4625 & .0375 & .0375 \\ .0375 & .0375 & .4625 & .0375 \\ .0375 & .0375 & .0375 & .8875 \\ .8875 & .4625 & .4625 & .0375 \end{pmatrix}^{-1} \frac{1-d}{n} \mathbf{1} = \begin{pmatrix} 17298 \\ 27466 \\ 51391 \\ 53843 \end{pmatrix} \propto \begin{pmatrix} .115 \\ .183 \\ .343 \\ .359 \end{pmatrix}$$

PageRank in relation to search engines

We now know how the algorithm assigns a 'popularity' value to each document. But how is this used to return actual search results?

First there is a web crawler that starts with a series of URLs that are fetched and visited. If the URL is not associated with an ID number it is assigned an ID number. When a new item is added to the database it is parsed and labeled with word occurrences. These are translated to

⁴This works because we can always get an inverse to a matrix M when we consider $(\mathbf{I} - |\lambda_1| |M|)^{-1}$ instead of $(\mathbf{I} - M)^{-1}$ but the proof of this is outside the scope of this class.)

'hits' which considers occurrence, location in document, size of font, capitalization and other features that could be used to infer importance and meaning. The links on the page (to other pages) are also recorded as is the anchor (what the hypertext actually is).

This data is then reduced to just the links and where they come from and point to which is then plugged into the PageRank algorithm. The additional information pulled out from the full webpages is then reduced to a lexicon that can be used by the searcher. This is combined with the PageRank scores to return on topic and relevant pages. There is also data being built up on the types of words (wordIDs) being built up that will then be used to select which documents might be returned as search results.

In practice, the search engine must:

- 1) parse the query
- 2) convert the words of the query to wordIDs
- 3) begin search through document list for each of the main words in query
- 4) scan through document list until there is a document that matches on all the search terms
- 5) compute the rank of that document for the specific query
- 6) Repeat until there are a certain number of results.
- 7) Sort the selected documents by rank and return the top k results.

As specified in Brin and Page 1998 the search results are ranked by a combination of PageRank and IR (relevance) score. IR scores are computed by converting the type of hits for each word into a type-weight and the number of hits converted to a count-weight. The dot product of these two becomes the IR score. For multiple word searches hits that are on topic and close together are weighted higher.

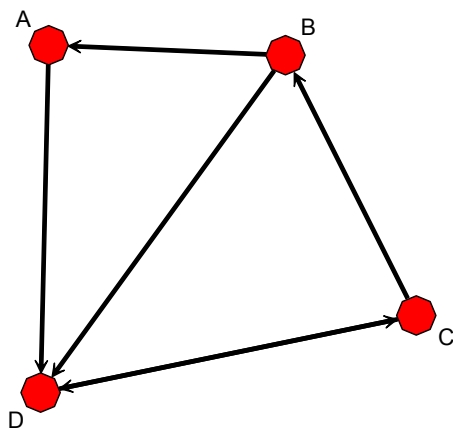


Figure 2: Toy example of a potential web graph. Each edge represents a hyperlink with the edge pointing toward the page you'd end up on if you clicked on the hyperlink.

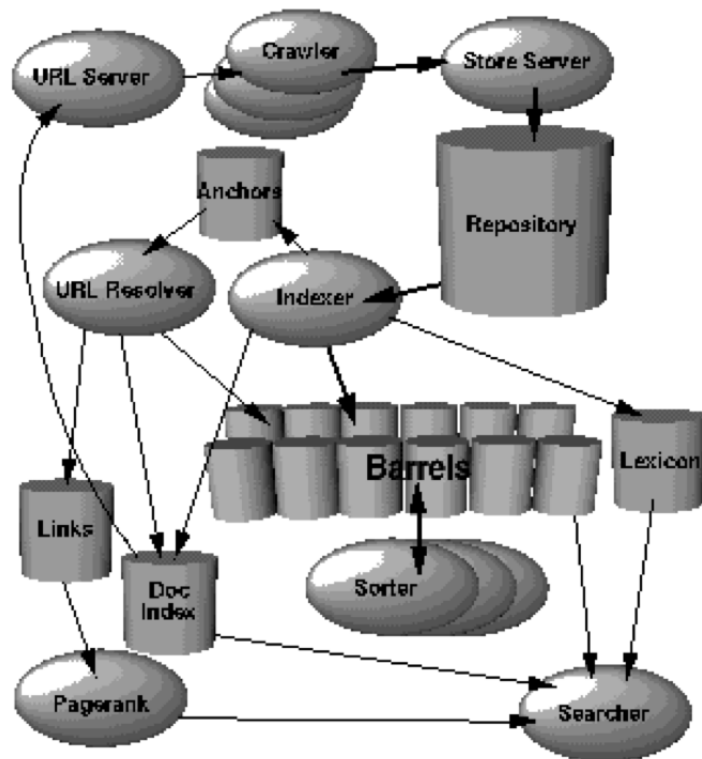


Figure 3: High level google architecture circa 1998. From Brin and Page paper.