An Overview of PageRank

Cassandra Overney, Adam Selker

April 2020

Introduction

The web is massive! In the beginning of 2004, it was estimated to contain over 10 billion pages. As of 2016, Google estimates that we have more than 60 trillion web pages, and that number continues to grow dramatically. New pages are constantly being added, and old ones are being updated.

In order manage and search through the web, we need a way to store information about each web page. Before Google, web information retrieval was not automated. Yahoo started as a manually curated catalogue of links. Today's web search engines (e.g. Google and Bing) use web crawlers to constantly scrape information, including which web pages link to which others.

Though the information from web crawling is a lot more concise than the actual web, it is still much too large to manually parse through. Due to the massive size of the web, a search phrase can result in countless matches. A user wants the most important pages that match the query, so an algorithm to sort pages by importance is quite useful. PageRank is one algorithm that does this.

PageRank views the web as a directed graph. Nodes are web pages, edges are links. If page a links to page b, then there will be an edge from a to b in the graph. PageRank uses links as recommendations, so a page with many links to it is viewed as more important. However, not only the volume of links matters but also the status of the recommender, and the total number of recommendations that it made (i.e. outbound links from the page). If a page that does not link to many other pages, then its recommendation carries more weight.

Equivalently, we can imagine a number of surfers browsing the web. They start on a random page, and click links at random. Sometimes, instead of clicking a link, they "teleport" to a random other webpage. Eventually, the distribution of surfers will stabilize. The importance of each page is how many surfers are on it, on average, after the distribution has stabilized.

PageRank is not only used in web search engines but also in a diverse range of other applications, such as recommendation systems, protein interaction networks, and production networks. Some interesting applications that caught our eye include using PageRank to detect anomalies in the movements of older adults, predict road and foot traffic in urban spaces, and evaluate commits on GitHub repositories.

The Mechanics of PageRank

PageRank defines the importance of a page as the sum of the weights of inbound links, plus a constant. The weight of a link is the importance of the page it comes from, divided by the number of outbound links from that site.

Let's describe this with matrices. We have the graph adjacency matrix A, of size n by n. The element $A_{i,j}$ is 1 if page i links to page j, and 0 otherwise. Our first step is to normalize the matrix, so that each row sums to 1, i.e. the weight of all of the outbound links from a page always sums to 1. This also means we have to deal with pages which link to nobody; we edit them to link to everybody.

$$M_{i,j} = \begin{cases} A_{i,j} = 1, & \frac{1}{\sum_k A_{i,k}} \\ \forall k\ A_{i,k} = 0, & \frac{1}{n} \\ \text{otherwise}, & 0 \end{cases}$$

Next, let's define our "damping factor" d. This is the chance that a hypothetical surfer clicks on a link, rather than teleporting. It's usually around 0.85 in practice. Also, just for convenience $\mathbf{1}$ is the column vector of n ones, and $\boldsymbol{I}$ is an identity matrix of size n by n.

We're now ready to set up the iterative "power method" for approximating PageRank. We start with a uniform $R(0) = \frac{1}{n}\mathbf{1}$, and iterate.

$$R(t+1) = dMR(t) + \frac{1-d}{n}\mathbf{1}$$

The first part of the right side (before the +) corresponds to the surfers who follow links; the second part corresponds to surfers who randomly teleport. After enough iterations (i.e. a large enough t), $R(t)$ should approach a steady state, where $R(t+1) \approx R(t)$. In practice, this seems to take $O(log(n))$ iterations. Each iteration contains one matrix product, taking $O(n^3)$ time, or a bit less if one uses a smarter algorithm.

We can also find the steady state of this series algebraically, by setting $R(t+1) = R(t)$:

$$\begin{aligned} R(t+1) &= R(t) \\ R(t) &= dMR(t) + \frac{1-d}{n}\mathbf{1} \\ R(t) - dMR(t) &= \frac{1-d}{n}\mathbf{1} \\ (\boldsymbol{I} - dM)R(t) &= \frac{1-d}{n}\mathbf{1} \\ R(t) &= (\boldsymbol{I} - dM)^{-1}\frac{1-d}{n}\mathbf{1} \end{aligned}$$

Unlike the power method, this gives us a perfect answer in a single step. How long does this step take? Like matrix multiplication, matrix inversion takes $O(n^3)$ time naively, or a bit less with a clever algorithm. Unlike the power method, we only do this once, so we don't have the extra factor of $log(n)$. However, we lose an important optimization: sparse matrices. The $O(n^3)$-ish runtime of matrix multiplication is true of dense matrices, where many of the elements are nonzero. However, the adjacency matrix is almost all zeroes, since most pages do not link to most other pages. Multiplying a dense matrix by a sparse matrix can be done more quickly (roughly quadratic with the number of nonzero elements). Sparse matrices also use less memory. So, in practice, the power method is almost always the best choice.

Let's take another look at the output of this algorithm. We find a vector R for which $MR = R$. This means that R is a right eigenvector of M, with an eigenvalue of 1. This is the principle eigenvector of M, since M is a stochastic matrix and cannot have an eigenvalue larger than 1. So, any eigenvalue-finding algorithm will work as a PageRank algorithm! In practice, these are slower and more resource-intensive than the power method.

There are several methods that can accelerate the power method even more. The *adaptive power method* takes advantage of the fact that most pages' importance values converge fairly quickly. Instead of having to recalculate every single ranking, it locks pages' weights if they converge within a certain tolerance.

Extrapolation involves approximating the subdominant eigenvector and subtracting it from the current PageRank vector, propelling the power method forward by cutting down the time it takes for the subdominant eigenvector to converge to 0. Since extrapolating the subdominant eigenvector requires pre-computing future iterations of the power method, it is typically only applied periodically (e.g. every 20 iterations of the power method), and can give modest speedups to the power method. For more speedup, *quadratic extrapolation* considers the third highest eigenvector, and can reduce the PageRank computation time by 50-300%.

Finally, *aggregation* clusters high-level web pages where lots of other pages sit (e.g. university web pages) into "hosts". This yields a simpler "host graph", where nodes are hosts rather than single pages. We can apply the PageRank algorithm to the host graph, and then separately within each host. From there, we can approximate the global PageRank vector by multiplying the page rank probabilities together.

Implementation

We implemented several different ways of determining PageRank. We obtained example graphs from the Stanford Large Network Dataset Collection and selected 3 applications: twitter social networks, a small section of Google's web graph, and a network of Wikipedia hyperlinks. Our GitHub repo can be found here. *create_network.py* contains helper functions that convert edge data into a Python NetworkX graph and visualizes it. *power_method.py* has all of the functions that determine the ranking of nodes in a NetworkX graph.

Our first PageRank implementation involved pre-computing the google matrix from the row normalized adjacency matrix. The google matrix is a stochastic, aperiodic, and primitive version of the adjacency matrix. We obtain the google matrix by first replacing all the 0 row vectors with a uniform vector of value $1/n$, n being the number of nodes in the input graph. This stochastic adjustment takes care of the dangling nodes, or nodes that do not point to any other nodes. In other words, instead of being stuck at a dangling node, a random surfer can jump to another node, removing unnecessary sinks in the graph.

We can make another minor modification to the adjacency matrix, which involves adding in the possibility for a random surfer to teleport to another node instead of just following the directed edges in the graph. The teleportation matrix is a uniform $n \times n$ matrix with value $(1-\alpha)/n$. α is a number between 0 and 1 that determines the probability that a random surfer will follow the edges of a graph, which means that $1-\alpha$ is the probability that the surfer would ignore the edges and teleport. (α is equivalent to d from the previous section.)

We can combine the adjacency and teleportation matrices together by multiplying the adjacency matrix by α and then adding it to the teleportation matrix. The resulting matrix is known as the google matrix. After pre-computing the google matrix, we run it through the power method with a starting uniform rank vector, π_0, with a value of $1/n$. We then iteratively set the rank vector as the product of the previous vector and the google matrix until the rank vector converges to an ϵ convergence tolerance.

Pre-computing the google matrix works well for the twitter social network data since the data consists of 973 smaller networks, each with no more than 300 nodes. However, since the google matrix is not sparse, a larger graph will quickly consume memory. When we tried this approach on the Google web graph, we ran out of RAM. Even when we had enough memory, the power method would take an exceedingly long time to run because we kept doing matrix operations on a huge and dense google matrix.

In response to the memory and time issues from our first implementation, we wrote another power method that utilizes the sparsity of the original adjacency matrix. Instead of pre-computing the google matrix, we row-normalized the adjacency matrix and calculated a binary vector of length n with a value of 1 at position i if the ith node was a dangling node and 0 otherwise. During the iterative process of the power method, we complete the following steps:

1. Compute the new rank vector when just considering normal node-to-node transitions, which is just α multiplied by the previous rank vector and the adjacency matrix

2. Determine the dangling vector where all of the dangling nodes have a value of $\alpha/n \times$ their previous ranks. All other values in the dangling vector are 0

3. Combine the likelihood of following connections (i.e. the sum of steps 1 and 2) and the likelihood of teleportation, a uniform, $1 \times n$, vector with value $(1-\alpha)/n$.

Using a sparse matrix saves a lot of time and memory, especially for larger graphs.

Our last PageRank implementation uses Numpy's built in eigenvalue function to find the principal eigenvector. It finds all of the eigenvectors and eigenvalues, so it is slower than the other two methods.

Because the Twitter graphs were fairly small, we were able to run all three methods on them. We analyzed the first 50 networks with an α of 0.5 and an ϵ of 1e-8. The number of nodes ranged from 13 to 226. The average time for the dense power method was 0.022 seconds; the average time for the sparse power method was 0.009 seconds; and the average time for the Numpy method was 0.057 seconds. In general, the sparse power method takes the least amount of time, followed by the dense power method, and then the Numpy method. Fig. 1 shows 3 examples of twitter networks where nodes are colored by their rankings. The green and yellow nodes tend to have more arrows pointing to them and, as a result, a higher ranking.

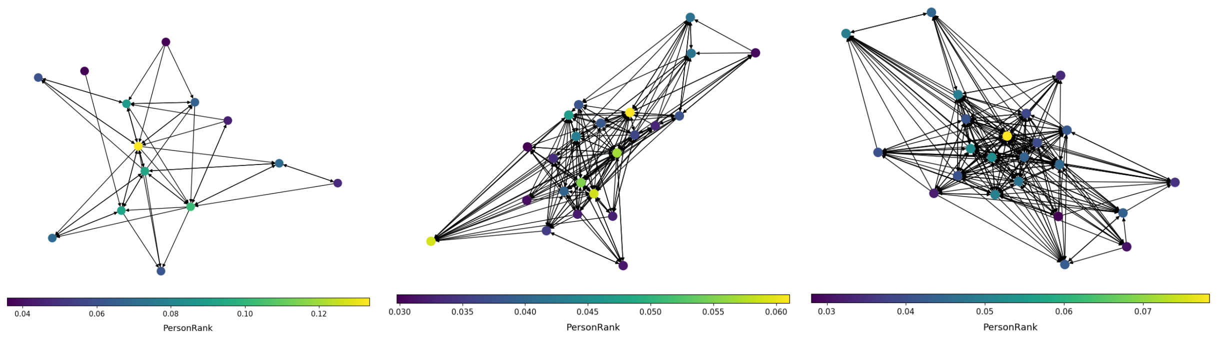

Figure 1: Networks 101903164 (left), 101859065 (middle), and 107172099 (right)

The other two applications were too large for the pre-computing and Numpy methods. The sparse PageRank algorithm took around 20 seconds to run for the google web graph, which has 875,713 nodes and 5,105,039 edges. The Wikipedia hyperlink network is even larger with 1,791,489 nodes and 28,511,807 edges and took around 200 seconds to analyze. The Wikipedia article names with the highest rankings, normalized by the largest, are:

1. United States, 100%
2. France, 48.7%
3. Canada, 31.9%
4. United Kingdom, 30.8%
5. Germany, 29.3%
6. Departments of France, 26.2%
7. Communes of France, 25.0%
8. Gmina, 23.6%
9. Village, 23.5%
10. Australia, 23.4%

Most of the highly ranked Wikipedia articles are associated with countries, which tend to be massive pages where many other articles link to. Though, the emphasis on France is quite surprising. Even Gmina is related to a country since it represents the principal unit of Poland's administrative division. It is interesting how most of these countries have either English, French, or German as their dominant language. These three languages make up, by far, the most articles in Wikipedia. Even so, it is important to note that the rankings after the top 2 articles cover a pretty small range.

References

[1] Our GitHub Repo.

[2] "How many web pages are on the Internet presently?". Source.

This source was used to get the metric of how many web pages there were in 2016.

[3] Stanford Large Network Dataset Collection. Source.

This source was used to get network data to test our PageRank implementation.

[4] Langville, Amy N., and Carl D. Meyer. Google's PageRank and beyond: The science of search engine rankings. Princeton university press, 2011.

This book provided a lot of the mathematical background that we based our PageRank power method implementations on. It also talked about different ways to accelerate the power method.

[5] Gleich, David F. "PageRank beyond the Web." SIAM Review 57.3 (2015): 321-363. Source.

This article discusses many interesting PageRank applications beyond web search engines.