UNIVERSITAT DE BARCELONA

Numerical Linear Algebra

master in fundamental principles of data science

Project 3

PageRank implementations

Author
Vladislav Nikolov Vasilev

Faculty of Mathematics and Computer Science

Academic year 2021-2022

1 Introduction

In this project we are going to work with the PageRank algorithm. Given a web page network with n web pages and a link matrix G, the PageRank score x_k of the page k is defined as

$$x_k = \sum_{j \in L_k} \frac{x_j}{n_j}$$

where L_k are the webpages linking to page k and n_j is the number of outgoing links from page j. This can be expressed as a fixed point equation $x = Ax$, where $A \in \mathbb{R}^{n \times n}$ is a modified version of the G matrix. Thus, the solution is given by the eigenvector with eigenvalue $\lambda = 1$. However, this vector may not be unique if the network is disconnected, or might not exist if it has dangling nodes.

To address these issues, one can consider the matrix $M_m = (1 - m)A + mS$. In this expression, we have that m is the damping factor, which is recommended to be set around $m = 0.85$, and S is a matrix containing random noise, which usually is $S_{ij} = 1/n$. This matrix has the property of being column stochastic, which ensures that there will be one unique solution to the PageRank problem.

In order to compute the PageRank scores of the web pages, we shall consider the power method, which is an iterative method that computes the next value as $x_{k+1} = Mx_k$ until it converges to the eigenvector with eigenvalue $\lambda = 1$.

2 Implementation

We have implemented two versions of the power method, which can be found in the `page_rank.py` file. The first one is the most intuitive and straightforward one, whereas the second one is focused on memory efficiency. Since we are going to deal with problems in which the matrix G is sparse (just a few entries of the matrix will be 1), we have to take advantage in both cases of the sparsity of the matrix in order to accelerate the computations and save as much space as possible. Basically, this means that we are going to work with the coordinate format in order to avoid using full matrices. These algorithms will iterate until $\|x_{k+1} - x_k\|_\infty < \text{tol}$, where tol is a given tolerance.

The first implementation is the `power_method` function. This version consists in iterating $x_{k+1} = (1 - m)GDx_k + ez^t x_k$ until it converges. First, we compute the n_j values for each web page as the number of outgoing links from the page j. With this, we can easily compute $(1 - m)GD$ before starting the iterative method,

since this value will not change over time. Taking advantage of the sparsity of G, we can just iterate over the entries of G, which are in coordinate format, divide each one of them by the corresponding n_j value, and then multiply the result by $(1-m)$. Something important to notice here is that the result is stored in a vector rather than in a matrix, which saves a lot of space. After that, we create the vector z, which is defined as $1/n$ if $n_j = 0$, or as m/n if $n_j \neq 0$. Once that is done, we perform the power iteration until it converges. In it, we first iterate over the previously computed values and update the x_i pages, and finally we add to this result the dot product $z^t x_k$. Instead of computing first the ez^t matrix and then performing the dot product with the x_k vector, it is much faster and memory efficient to perform the previously mentioned dot product between the vectors (which will produce a real number) and take advantage of `numpy`'s broadcasting mechanisms to update the whole vector. This saves us the need to store the vector e, since it becomes redundant.

The second implementation can be seen in the `power_method_no_matrix` function. Here, the power iteration changes, and also the way that the data is computed. For instance, the vector z^t is not computed, and neither is the GA product. First, for each page we create a list of pages that link to it, which corresponds to L_j. Also, for each page we count how many outgoing links it has. Then, we perform the power iteration as it is specified in the problem's description. We iterate over the pages (in the previous implementation we iterated over the computed values instead of over the pages) and check in each case whether $n_j = 0$. If the condition is met, then the score vector is updated by adding x_j/n to each page x_i. Otherwise, we update the linked pages L_j by adding x_j/n_j to each one of them. A small difference with respect to the proposed implementation is that the inner loop inside the `else` statement has been replaced with a direct update by accessing the pages using `numpy`'s capabilities of accessing multiple elements at the same time. Finally, we multiply the score vector by $(1-m)$ and add m/n to each page x_i. This implementation may actually be slower than the previous one due to the fact that we have to check extra conditions in each iteration of the loop, but has the advantage of requiring less memory, which is especially useful in big problems.

3 Experimentation

We are going to test how the previously described implementations work using the `p2p-Gnutella30.mtx` matrix. This matrix represents a network made up of 36682 nodes and 88328 connections between the nodes, which is in fact much smaller than the total number possible connections, which is of the order of $36682^2 \approx 1.3456 \times 10^9$. Because the matrix is indeed sparse it is represented using the coordinates format.

We are going to fix the damping factor to $m = 0.85$ and are going to test different values of tolerance to see how the results change. For each experiment, we are going to compute how much time it takes to compute the PageRank score, how many iterations are needed, the relative error of the eigenvector, how many incorrect pages have been computed and the first incorrect page. In order to compute these three last values, we are going to set the result obtained with the lowest tolerance as the reference, since we know that it is going to be the closest one to the real solution of the problem. In order to compute the error, we are going to use the $\|\cdot\|_2$ norm.

The results can be seen in tables 1 and 2, which contain the results of the experiments for the first and second implementation, respectively. We can see that the first implementation is faster than the second one (approximately twice as fast). This is not surprising at all considering that the second method requires more operations in order to perform the power iteration. Therefore, we can see the trade-off between fast algorithms and algorithms that require less memory: we cannot have both of them at the same time, and usually using less memory requires more computing time.

Both methods require the same number of iterations to converge to the solution and have the same relative error for the same tolerance values. However, the number of incorrect pages and the first incorrect page change when we compare both of the methods with the same tolerance value. The most probable reason for this is how the power iteration is computed.

We can see that using small values of tolerance yields good results with a small relative error, less incorrectly ordered pages and the first incorrect page has a higher value (except for the tolerance 1e-10, which might be due to how the sorting algorithm works when there are multiple pages with the same score). However, the price to pay is that the number of iterations increases, which directly impacts the run time. It is clear that using high tolerance values is not recommended, because the obtained results are not very good (they get wrong the first page, which is actually the most important one). However, using very low tolerance values might also not be the best idea, because depending on the problem's size, the computation times might be even higher. Thus, using some of the intermediate tolerance values (1e-08 or 1e-07, for instance) might be a good idea, since the computation times are not very high and the results are quite accurate.

With this, we can conclude that the value of the tolerance plays a huge impact on the results. Higher values offer in general worse and less accurate results, while using lower values yields better results in exchange of higher computation times. Consequently, we have to carefully choose the tolerance value so that it is an intermediate value.

Tolerance	Time (s)	Iterations	Error	Num. incorrect pages	First incorrect page
1e-14	10.5376	74	0	0	0
1e-12	9.0869	60	2.9963e-12	20	7626
1e-10	7.1703	47	2.6186e-10	42	92
1e-08	4.9428	32	5.301e-08	102	956
1e-07	4.0683	27	3.3869e-07	1025	83
1e-06	3.1645	21	3.4679e-06	2691	83
1e-05	2.2866	15	3.8219e-05	11948	24
0.0001	1.2363	8	0.0006	20747	3
0.001	0.1979	1	0.0091	20980	0
0.01	0.1974	1	0.0091	20980	0
0.1	0.1936	1	0.0091	20980	0

Table 1: Results of the first implementation of the power method.

Tolerance	Time (s)	Iterations	Error	Num. incorrect pages	First incorrect page
1e-14	20.09	74	0	0	0
1e-12	15.8848	60	2.9963e-12	932	6653
1e-10	13.0007	47	2.6188e-10	918	92
1e-08	8.7209	32	5.301e-08	989	956
1e-07	7.2494	27	3.3869e-07	1240	83
1e-06	5.8946	21	3.4679e-06	3534	83
1e-05	4.1057	15	3.8219e-05	11967	24
0.0001	2.2238	8	0.0006	20718	3
0.001	0.3469	1	0.0091	20981	0
0.01	0.3691	1	0.0091	20981	0
0.1	0.36666	1	0.0091	20981	0

Table 2: Results of the second implementation of the power method.