This report corresponds to the evaluation of the results obtained in the third project of Numerical Linear Algebra. The report will not be very long, I will simply explain how I have implemented the delivered code, as there is no theoretical exercise to be delivered.

Before starting, I would like to comment how the delivery has been made. The corrector has received a *zip* file with this *pdf* file, corresponding to the evaluation of the obtained results and one python script, called *PageRank.py*, corresponding to the solution of the proposed problems.

1 PageRank implementations

The python script contains three functions and a main part. There are two functions which serve to create the matrices D and A. To create the sparse matrix D, the function *create_D*, for a given link matrix G, computes the out-degree values, n_j of a page j and subsequently creates the diagonal matrix $D = \text{diag}(d_{11}, \cdots, d_{nn})$ where $d_{jj} = \frac{1}{n_j}$ if $n_j \neq 0$ and $d_{jj} = 0$ otherwise. Once we have the sparse matrix D, we can compute the matrix $A = GD$. In order to implement this, we have made a python function called *create_A* that performs this multiplication. Let us focus on the required exercises. We have implemented two functions for each exercise, *PR_store* and *PR_without_store*.

Let us start with the explanation of the first exercise, the computation of the PageRank vector (PR vector from now on) of the matrix $M_m := (1-m)A + mS$. The first approach proposed to solve this is to use the power method (adapted to PR computation) with the possibility of storing matrices. The algorithm reduces to iterate $x_{k+1} = (1-m)A + ez^t x_k$ until $||x_k - x_{k+1}||_\infty < \text{tol}$.[1] Remember that for each iterative algorithm a start point is needed, thus, we propose $x_0 = (1/n, \cdots, 1/n)$. When the iterative algorithm is finished, the function returns the obtained vector but normalized.

On the other hand, the function *PR_without_store* computes the PR vector of M_m using the power method but without storing any matrix. The algorithm reduces to an iterative structure given by the teacher.

The difficulties presented by both methods are the following: the computation of the vector z in the first approach and the computation of $L_k = \{\text{webpages with link to page } k\}$ and $n_j = \{\text{number of outgoing links from page j}\}$ in the second approach. Let us start with how the computation of z has been made. Remember that $z = (z_1, \cdots, z_n)^t$ is the vector given by

$$z_j = \begin{cases} m/n \text{ if the column } j \text{ of the matrix } A \text{ contains non-zero elements.} \\ 1/n \text{ otherwise.} \end{cases}$$

Therefore, we need to know if the column j of the matrix A, that is store in a sparse way, contains non-zero elements. Notice that if we have a sparse matrix A, the command $A.indices$ will return an array mapping each non-zero element to its column in the sparse matrix. Let us see an example.

Example 1.1. *Assume that we have the following sparse matrix A:*

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

[1] The project proposes to fix m to 0.15.

Then, A.indices = [0,2,2,3,4].

Thus, the command *np.unique(A.indices)* will return the columns of the matrix A with non-zero elements.[2] Thus, since we have access to the columns of the matrix with non-zero elements, the vector z is now easy to compute.

The last problem is to calculate the numbers L_k and n_j mentioned above. In order to compute L_k we have to understand the command *indptr*. Firstly, we need to know what the command *data* do. For a given sparse matrix A, $A.data$ is an array containing all the non-zero elements of the sparse matrix. In the previous example, $A.data$=[1,3,2,1,1]. Now, the command *indptr* maps the elements of data and indices to the rows of the sparse matrix in the following way: for row i, $[indptr[i] : indptr[i+1]]$ returns the indices of elements to take from data and indices corresponding to row i. So suppose $indptr[i] = k$ and $indptr[i+1] = l$, the data corresponding to row i would be $data[k : l]$ at columns $indices[k : l]$[3]. In the previous example, $A.indptr$=[0,0,2,2,5,5]. Let us look at an outline to clarify ideas.

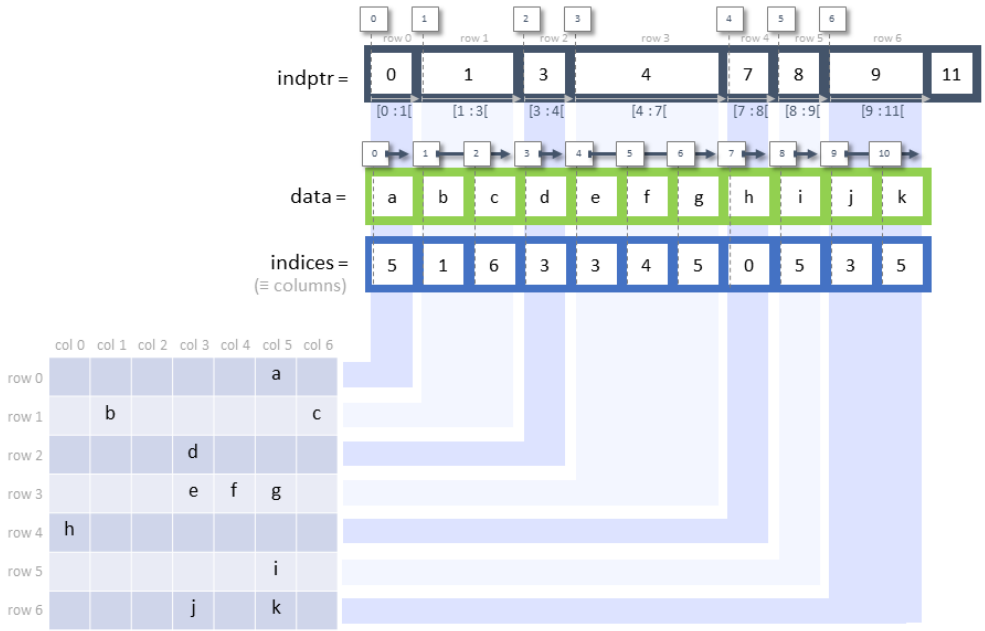

Figure 1: Example of the functions data, indices and indptr.

Then, since the values of our matrix represents the links, we have obtained an easy way to compute L_k. Moreover, the value of n_j is the length of L_k.

2 Results

We have set the tolerance to 10^{-12} and $m = 0.15$. The result is very similar for both methods, indeed, the difference between them is of the order of 10^{-11}. The biggest difference between both methods is the computational time. For the first method, storing matrices, the computational

[2] We have to add the *np.unique* to avoid repetitions.

[3] If the reader desires to find more information about this, I recommend the next forum https://stackoverflow.com/questions/52299420/scipy-csr-matrix-understand-indptr

time is around 0.03 and 0.06 seconds, while for the second one, without storing matrices, the computational time is around 9.3 and 11 seconds. Notice that the "price" that we have paid for not storing any matrix is the computational time!