Project 3 NLA: PageRank implementations

Theodoros Lambrou

This report serves as the explanation on how the results were reached for this project. In the ZIP file I supplied the python file which contains the relevant code.

The python script has 3 functions and the main part; 2 functions for creating 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 $D = diag(d_{11}, \cdots, d_{nn})$ where $d_{jj} = \frac{1}{n_j}$, if $n_j \neq 0$ and $d_{jj} = 0$ otherwise.

After obtaining the sparse matrix D, we can calculate the matrix *A* = *GD*. To achieve this, I created a python function called `create_A` that performs this multiplication. For the required exercises, I have implemented two functions for each exercise, `PR_store` and `PR_without_store`.

For the first exercise, I did the computation of the PageRank vector 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 $||x_k - x_{k+1}||_\infty < tol$. (The project proposes to fix m to 0.15). A reminder that for each iterative algorithm a start point is needed, hence, I 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` calculates the PR vector of M_m using the power method but without storing any matrix. The algorithm reduces to an iterative structure as given in the class.

There are difficulties found in both methods: the computation of the vector *z* in the 1st approach and the computation of $L_k = \{$webpages with link to page k$\}$ and $n_j = \{$number of outgoing links from page j$\}$ in the 2nd approach.

$z = (z_1, \cdots, z_n)^t$ is the vector given by

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

Hence, it is necessary to know if the column *j* of the matrix *A*, that is stored in a sparse way, contains non-zero elements. 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.

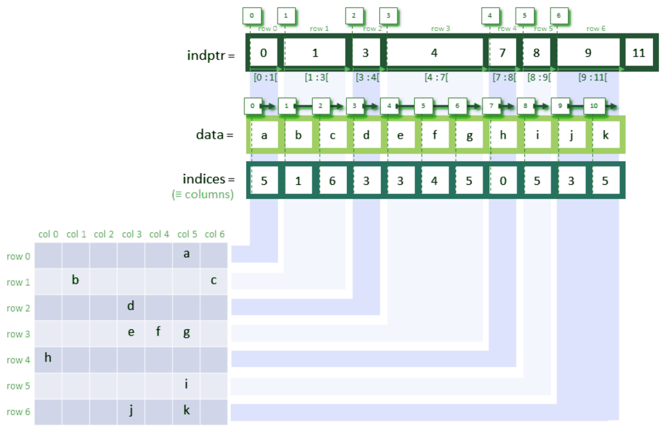
For example if we have a sparse matrix A like the following,

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

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

Hence, the command *np.unique(A.indices)* will give the columns of the matrix *A* with non-zero elements (We have to add the *np.unique* to avoid repetitions.)
Since we have access to the columns of the matrix with non-zero elements, the vector z is easier to calculate now.

The last problem is to calculate the numbers L_k and n_j mentioned above. To calculate L_k it is required understand the command *indptr*. First of all, it is necessary to know what the command *data* does. 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]. In the previous example, *A.indptr* = *[0,0,2,2,5,5]*. In the example image below, the functions data, indices and indoor can be seen.



So, since the values of our matrix represent the links, we have found an easy way to compute L_k. Moreover, the value of n_j is the length of L_k.

<u>Results</u>

The tolerance is set to 10^{-12} and $m = 0.15$. The result is very similar for both methods — the difference between them is in the order of 10^{-11}. The only significant difference between the 2 methods is the computational time. For the first method, storing matrices, the computational time is about 0.03 and 0.06 seconds. For the second one, without storing matrices, the computational time is about 9.3 and 11 seconds. Hence, storing matrices, saves us computational time.