Project 3: Page Rank

Lorenzo Vigo

Numeric Linear Algebra: 17th December 2021

1. Implementations

There is not much to comment about the *store-using implementation.* All we need to do is to implement the iterative method, taking into account that both current and previous iteration values have to be kept.

The execution time for this method lies around 20 milliseconds. It is really fast, but as explained in the statement, it requires large storage capacity if the dataset is big enough.

For that reason, we are proposed to implement an *implementation without store.* In that sense, no matrices are used in the operations of the iterative process: only vectors. That piece of code was handed in the statement, even though some variable names had to be changed.

In order to declare the L array that includes all L_k for all k pages, as mentioned in the statement, we are using the parameter `indptr` of Scipy's sparse `csc_matrix`.

The way `indptr` is the following: the `data` attribute of a sparse matrix includes its non-zero values. For example, let's consider:

$$\texttt{mat} := \begin{pmatrix} 0 & 0 & 0 \\ 5 & 0 & 0 \\ 0 & 1 & 7 \\ 0 & 0 & 0 \\ 0 & 0 & 8 \end{pmatrix}$$

$$\texttt{mat.data} = [5, 1, 7, 8]$$

The `indices` property of `mat` shows in which column each one of the values in `data` is. In our example:

$$\texttt{mat.indices} = [0, 1, 2, 2]$$

Both `indices` and `data` will have the same length: the number of non-zero values in the matrix. `indptr`'s length will be the number of the rows of `mat` and it maps the values to the rows in the matrix in the following way: the values included in a row i are the values found in the following positions of the array `data`: [`indptr[i]`, `indptr[i+1]`). If this interval is invalid, no values are included in that row. In our example,

$$\texttt{mat.indptr} = [0, 0, 1, 3, 3, 4]$$

For row 0, we get the interval $[0, 0)$, so no values are included there. For row 1, we get the interval $[0, 1)$, therefore the value `data[0]` is included in this row at column `indices[0]`. For row 2, we get the interval $[1, 3)$, so we know it will contain two values: `data[1]` and `data[2]`. Same result for row 3 as in row 0. And last value will be included in last row, as the obtained interval is $[3, 4)$. This way we have a mapping between non-zero values and their positions in the matrix.

As the values in our matrix represent the links, this way we will retrieve for each page k, the webpages with links to k. The value of n_j is easy to obtain by computing the length of each L_k.

This non-store version of page rank is considerably slower: the execution will be a little over 10 seconds long. Anyways, it may be worth the wait if the other implementation is not executable due to storage requirements.

The results given by both implementations are equal (with respect to machine precision). We can comment two final aspects common to both implementations: to build D, we are using `np.divide` in order to compute the inverse of each one of the values that should go in the diagonal.

The `out` parameter of this method defines the shape of the output, while the `where` parameter defines a condition to be fulfilled in order to perform the operation. We will use this last parameter in our advantage, in order to avoid dividing by zero. Where the operation is not done, default value 0 is inserted. We use a Scipy's sparse `diags` (diagonal) matrix in order to represent D.

Finally, it should be highlighted that we are normalizing all vectors in order to avoid divergence throughout the iterative process.

2. Influence of parameters

The *tolerance* parameter defines how much progress must be done from one iteration to another in order for the algorithm to consider it is worth to keep going. This kind of parameter is commonly known as *stopping criteria*. Decreasing the tolerance makes the algorithm more demanding: therefore, more iterations will be done and execution times will be higher. In exchange, more precision will be obtained. Our experiments showed great precision, so we will not analyze the effects of changing the tolerance for our algorithms.

The *damping factor* ($m \in [0, 1]$) represents the random behavior users may have. The lower m is, the more random user clicks will be considered. Our original experiments were performed with $m = 0.15$. We may change this value and see which effects in execution times and results it causes.

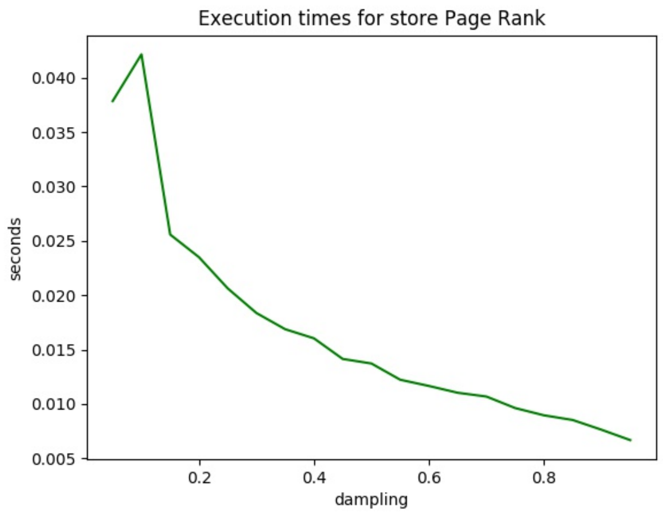

Figure 1: Execution times for Store Page Rank for different damping values

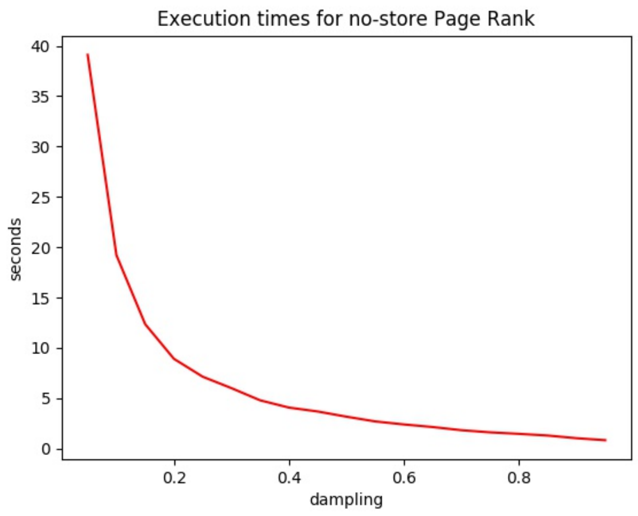

Figure 2: Execution times for Non-Store Page Rank for different damping values

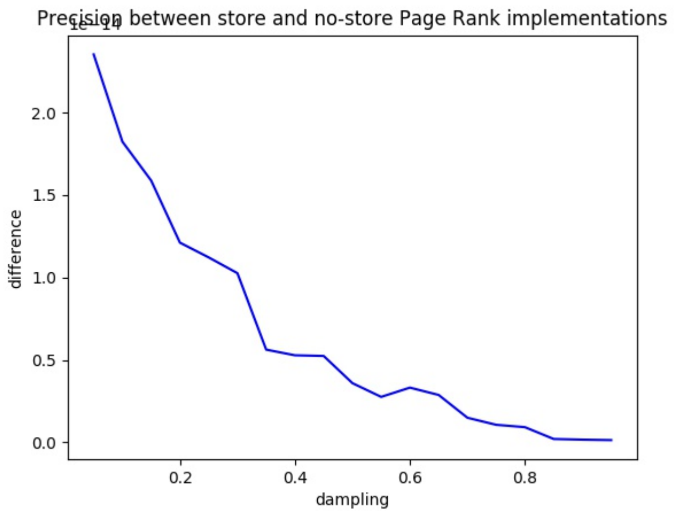

Figure 3: Difference between the solutions given by both implementations for different damping values

We can see that higher damping values lead to faster executions in both implementations. In non-store implementation, the decrease in execution times is really considerable: from 40 seconds to a couple of seconds. Also, higher damping values provoke more similar results between both implementations. This is logical, as higher values of m imply less randomness in our algorithm. In any case, the improvement is not outstanding, as the difference is always lower than $2e^{-14}$.