Numerical Linear Algebra

Master on Fundamentals of Data science
Universitat de Barcelona

Project 3
PageRank Implementations

Blai Ras

December 21, 2020

Contents

1 Results

The Page Rank algorithm can be implemented in multiple ways. In class, we are introduced to 2 different types of implementations: as a linear system and with the power method. The second one can be programmed in two different ways: storing the matrices and without.

In this section we will be discussing the different outputs of each method with two examples of sparse matrices: `p2p-Gnutella30` (36682 x 36682) and `p2p-Gnutella31` (62586 x 62586), extracted from the University of Florida Sparse Matrix Collection.

1.1 Linear system

Trying to find the PR vector of a big sparse matrix is a very bad idea. We want:

$$\hat{Mm} = (1-m)A + \hat{S}$$

$$\text{where } A = G \cdot D$$

$$\text{where } D \text{ is a diagonal matrix as} = \begin{cases} \frac{1}{nj} & \text{if } n_j \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{and } n_j = \sum_i g_{ij}$$

Isolating x we have:

$$(Id - (1-m)GD)x = e\hat{z}^T x$$

This can be done with the `scipy.sparse.linalg.spsolve` method if we take in consideration that $\sum x_1 = 1$ and therefore b is an array of dimension (n x 1) of ones.

The computation of the PR vector using this method lasted for one minute for the first matrix and 6.379 minutes for the second one, using a dampling factor of 0.15. In conclusion, too much computational cost.

1.2 Power method storing matrices

This time we have again:

$$\hat{M}m = (1 - m)A + \hat{S}$$

But

$$\hat{S} = e\hat{z}^T \text{ where } \hat{z}_j = \begin{cases} \frac{m}{n} & \text{if the column j of A is non-zero} \\ \frac{1}{n} & \text{otherwise} \end{cases}$$

In this case the operation that takes the most computational time is the creation of the z_j array. This are my results with a dampling factor of 0.15:

Matrix	**Tolerance**	**Time (s)**
p2p-Gnutella30	1e-5	20.73
p2p-Gnutella30	1e-15	21.12
p2p-Gnutella31	1e-5	61.28
p2p-Gnutella31	1e-15	61.91

We can see that the tolerance does not affect much the computational time. Let's try changing the dampling factor to 0.35:

Matrix	**Tolerance**	**Time (s)**
p2p-Gnutella30	1e-5	20.73
p2p-Gnutella30	1e-15	21.12

Again, the two times are very similar. I avoid the computation for the even bigger `p2p-Gnutella31` matrix.

1.3 Power method without storing matrices

Same idea but this time we are working with indexes instead of storing the whole matrix. This means considering:

$$j \text{ such that } n_j \neq 0 \Rightarrow M_{ij} \begin{cases} 0 & \text{if } g_{ij} = 0 \\ (1-m) & \frac{1}{n_j}\text{if } g_{ij} = 1 \end{cases}$$

$$g_{ij} = 0 \Longleftrightarrow \nexists \text{ link } i \to j \Longleftrightarrow i \not\supset L_j$$

My results, with a dampling factor of 0.15 are:

Matrix	Tolerance	Time (s)
p2p-Gnutella30	1e-5	4
p2p-Gnutella30	1e-15	4.08
p2p-Gnutella31	1e-5	8.62
p2p-Gnutella31	1e-15	8.61

We definitely see a big time improvement without the storage of the matrices. I see a bit of improvement if I lower the tolerance. Let's do more tests:

Matrix	Tolerance	Time (s)
p2p-Gnutella30	1e-4	4.15
p2p-Gnutella30	1e-25	4.06
p2p-Gnutella30	1e-30	4.02

So, by lowering a bit the tolerance this method gets a little bit faster. That can be explain because it's using the sparse matrix properties. The other method, the one where we store the matrices, what really takes time is the creation of the z_j vector, and this is only created once. Once created, the method ends quickly.

You can see a time comparison in the figure 1.1.

1.4 Convergence

Computing the eigenvalues of A I get $|\lambda_1| = 0.66$ and $|\lambda_2| = 0.49$. This means that their division ($|\lambda_2|/|\lambda_1|$) is equal to 0.73, which is smaller than 1-0.15. In conclusion, it converges but it is not as fast as it could be, because the method converges slowly if there is an eigenvalue close in magnitude to the dominant eigenvalue.

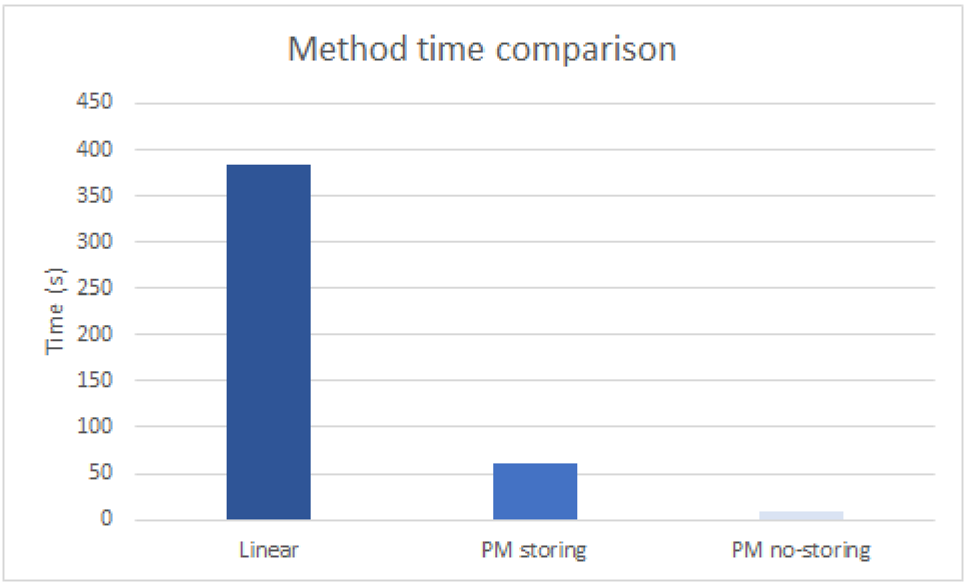

Figure 1.1: Comparison of the execution time of all 3 methods with a tolerance of 1e-5 and dampling factor of 0.15