Project 3: PageRank implementations
NLA

Clàudia Valverde

December 22, 2024

The goal of this project is to code two different implementations of the PageRank algorithm.

1 The Page Rank Algorithm

PageRank is an algorithm developed by Larry Page and Sergey Brin, founders of Google, to rank web pages based on their importance. It works by considering the link structure of the web, where each page's rank is determined by the number and quality of incoming links. The basic idea is that a page is important if it is linked to by many other important pages.

Given a webpage network with n webpages and a link matrix G (which defines a directed graph), the PageRank vector $\mathbf{x}$ can be defined as the solution to the following fixed point equation:

$$\mathbf{x} = A\mathbf{x},$$

where $A \in \mathbb{R}^{n\times n}$ is the transition matrix of the network. If the web network does not contain dangling nodes (i.e., nodes with no outgoing links), then the matrix A is column-stochastic. In this case, $1 \in \text{Spec}(A)$, meaning that 1 is an eigenvalue of A. If the network is connected and the eigenvector corresponding to eigenvalue 1 is unique, this eigenvector is the so-called PageRank (PR) vector.

However, there are two issues to address:

1. For disconnected networks, the PageRank vector may not be unique.
2. If the network has dangling nodes (nodes with no outgoing links), the matrix A is column substochastic, and it does not have an eigenvalue of 1.

To resolve these issues, we modify the transition matrix to include a teleportation vector:

$$M_m = (1-m)A + mS,$$

where:

- A is the original link matrix (which may be column substochastic),
- m is the teleportation factor, typically set to 0.85, but in our practical cas 0.15.
- S is the teleportation matrix, which handles the teleportation effect (e.g., when a user jumps to a random page, as in the case of dangling nodes).

The matrix M_m is column stochastic and guarantees the existence of a unique PageRank vector.

2 Function Explanation

We implemented two versions of the PageRank algorithm: one that explicitly stores the matrix M_m and one that avoids matrix storage by operating directly on the graph structure.

2.1 With Matrix Storage

In the first method, the graph is normalized and stored as a sparse matrix M_m. The algorithm then iteratively updates the PageRank vector $\mathbf{x}$ using matrix-vector multiplication. This method uses the normalized graph matrix M_m, which is constructed by scaling the adjacency matrix G with the inverse of the out-degree of each node. It handles dangling nodes by adding a teleportation vector $\mathbf{z}$, which is computed using the out-degree information of the graph.

The function 'compute_pagerank' performs the following steps:

1. Normalizes the adjacency matrix G to create a column-stochastic matrix A.
2. Adds the teleportation matrix S to handle dangling nodes and form the matrix M_m.
3. Uses the power iteration method to compute the PageRank vector $\mathbf{x}$ iteratively until convergence.

2.2 Without Matrix Storage

The second method avoids explicitly constructing the full matrix M_m. Instead, it operates directly on the adjacency list representation of the graph. This method iterates over the non-zero entries of the graph, distributing the rank of each node to its neighbors based on the adjacency structure. It still accounts for dangling nodes using a teleportation vector $\mathbf{z}$, but it does not require matrix storage, making it more memory efficient.

The function 'compute_pagerank_no_matrix' does the following:

1. Computes the adjacency list of the graph and determines the out-degree for each node.
2. Iterates over the nodes, updating the PageRank vector based on their incoming links.
3. Handles dangling nodes by distributing their rank uniformly across all nodes and adding the teleportation vector $\mathbf{z}$.

3 Comparison of Methods

The primary difference between the two methods is in how the graph structure is represented. The matrix-based method stores the entire normalized graph as a matrix M_m and uses matrix-vector multiplication for the power iteration. This method is more straightforward but requires more memory, especially for large sparse graphs. On the other hand, the adjacency list method avoids storing the matrix M_m and directly operates on the graph structure, making it more memory-efficient at the cost of slightly increased computation time due to the need for explicit iteration over the graph's edges.

Both methods converge to the same result, but the matrix storage method is often faster for dense graphs, while the adjacency list method may be preferable for very large sparse graphs.

Lets see this in a practical expercise using the matrix `p2pGnutella30.mtx`

Ex 1. Compute the PR vector of M_m using the power method (adapted to PR computation). The algorithm reduces to iterating:

$$\mathbf{x}_{k+1} = (1-m)G_D\mathbf{x}_k + e_z^T\mathbf{x}_k$$

until

$$\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_\infty < \text{tol}.$$

Ex 2. Compute the PR vector of M_m using the power method without storing matrices.

We compute the PageRank (PR) vector of the matrix M_m using two different methods:

1. **With Matrix Storage**: The standard power method, where the matrix is explicitly stored.
2. **Without Matrix Storage**: The power method implemented without explicitly storing the matrix M_m, leveraging sparse matrices.

Both methods were run with the following parameters:

- **Tolerance**: 1×10^{-6}
- **Damping Factor (m)**: 0.15

The top 10 PageRank indices and values obtained from both methods are as follows:
Results for the Method with Matrix Storage:

- **Top 10 PageRank Indices**:

$$[31803, 31366, 24973, 9475, 29641, 12684, 19063, 31548, 36465, 33103]$$

- **Top 10 PageRank Values**:

$$\begin{bmatrix} 1.44 \times 10^{-3}, 1.33 \times 10^{-3}, 1.26 \times 10^{-3}, 1.12 \times 10^{-3}, 1.10 \times 10^{-3}, \\ 1.10 \times 10^{-3}, 9.64 \times 10^{-4}, 9.60 \times 10^{-4}, 9.44 \times 10^{-4}, 9.34 \times 10^{-4} \end{bmatrix}$$

Results for the Method without Matrix Storage:

- **Top 10 PageRank Indices**:

$$[31803, 31366, 24973, 9475, 29641, 12684, 19063, 31548, 36465, 33103]$$

$$\begin{bmatrix} 1.46 \times 10^{-3}, 1.34 \times 10^{-3}, 1.28 \times 10^{-3}, 1.13 \times 10^{-3}, 1.12 \times 10^{-3}, \\ 1.12 \times 10^{-3}, 9.77 \times 10^{-4}, 9.74 \times 10^{-4}, 9.57 \times 10^{-4}, 9.47 \times 10^{-4} \end{bmatrix}$$

Observations:

- Both methods give very similar results for the top 10 PageRank indices, with the differences in the PageRank values being relatively small.
- The slight differences in the values can be attributed to numerical approximations and the method of computation (explicit matrix storage vs. sparse matrix representation).
- The differences are within the tolerance range and are expected due to the iterative nature of the algorithm.

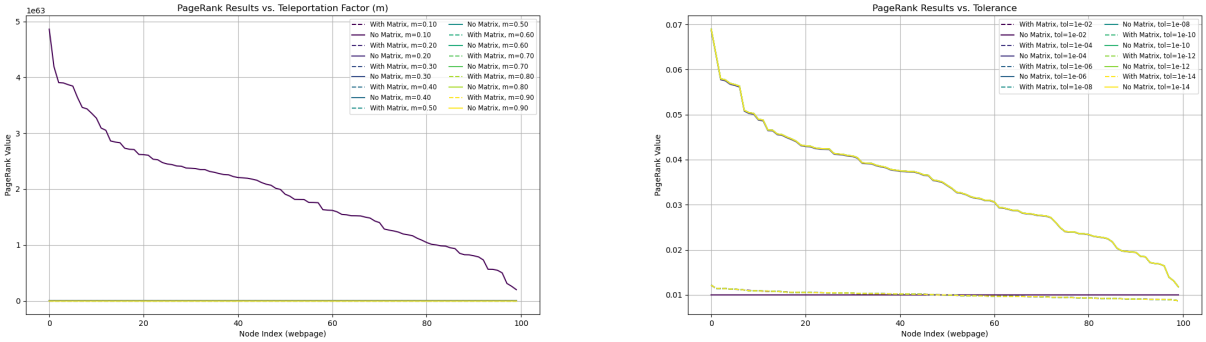

Figure 1: PageRank values change as the teleportation factor m varies (left) or tolerance vaires (right), with the two methods (with and without matrix storage).

We tested the algorithm with different teleportation factors (m values) and varying tolerances. For more details, see Figure (1). When varying the teleportation factor, we did not observe significant differences in the PageRank values. For m values greater than 0.1, the PageRank values tend to become very small, approaching zero. In contrast, when changing the tolerance, although the behavior remains relatively stable across different values, we observe a clear distinction between the two methods. For the method that stores the matrix, PageRank values tend to decrease as the node index (i.e., the webpage

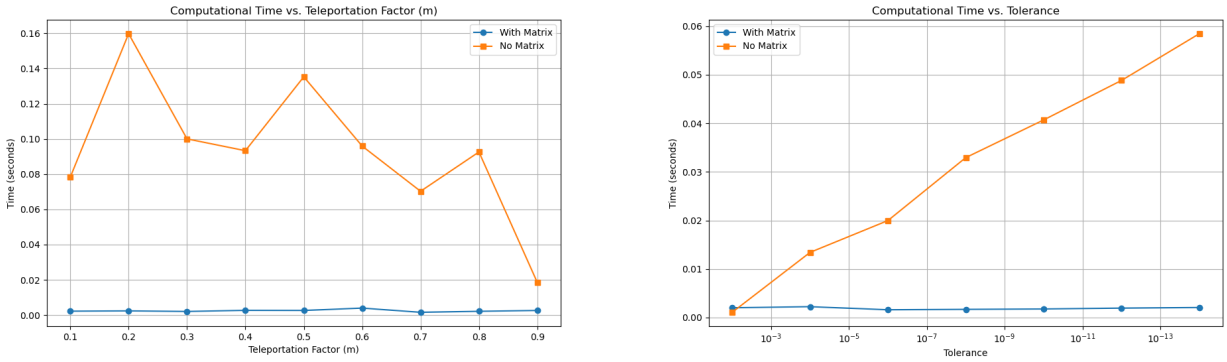

Figure 2: Computational time of both methods for different teleportation facotrs (left) or tolerances (right).

index in the adjacency matrix) increases. On the other hand, the method without matrix storage results in much smaller PageRank values even for lower node indices.

As shown in Figure 2, the method that avoids storing matrices takes longer to compute as the tolerance decreases, compared to the method that stores matrices. This is due to the fact that the second method (without matrix storage) is iterative in nature. Therefore, we can conclude that the trade-off for not storing matrices is not related to storage space, but rather to increased computational time.