
# Big Data

PageRank algorithm

Cyril Tsilefski

Program made in Python

Attached files:

*main.py*
*function_task1.py*
*function_task2.py*
*parameters1.py*
*parameters2.py*

# Contents



### Disclaimer

In this report, I adopt a writing convention:

- Filename are in *italic*
- Any value, variable, string, object ... is in **boldface**

Please note that the output files are quite big, I will provide the first ten results of each language in the report, the complete list of results will be available on the github page of this project in *project/data_out/*.

# 1 Introduction

## 2 Task 1

In Graph Theory, a graph is a structure made of vertices connected by edges. These graph can be directed (asymmetrically) and undirected (symmetrically). The PageRank method uses the graph theory to rank pages of the web by analyzing the hyperlinks between pages. In this part we will see how this algorithm works from a matrix approach using the following graph.

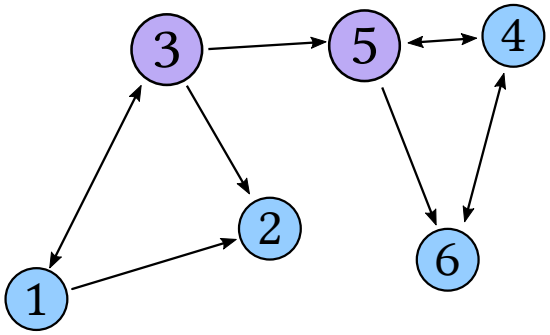


Figure 1: Graph used to experiment with the algorithm

The first mathematical object needed is the adjacency matrix. It is a matrix encoding the structure of the graph. This matrix is defined by:

$$A_{ij} = \begin{cases} 1 & \text{if } j \text{ point to } i \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The adjacency matrix of the considered graph is:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Since our graph is directed, this matrix is not symmetrical.

Once the adjacency matrix is build, we need the stochastic matrix:

$$S_{ij} = \begin{cases} A_{ij}/k_j^{\text{out}} & \text{if } k_j^{\text{out}} \neq 0 \\ 1/N & \text{otherwise} \end{cases} \quad \text{with } k_j^{\text{out}} = \sum_{i=1}^{N} A_{ij} \tag{2}$$

The out-degree $k_j^{\text{out}}$ corresponds to the number of vertices to leave the vertex $j$. An element $S_{ij}$ of the stochastic matrix corresponds to the probability of jumping from node $i$ to node $j$. The second part in the definition od $S$ is there is case we end up on a node without any escape ($k_{\text{out}} = 0$). In that case we have a probability $1/N$ of jumping into any node of the network.

The out-degree $k^{\text{out}}$ and the stochastic matrix $S$ of the considered graph are:

$$k^{\text{out}} = \begin{pmatrix} 2 \\ 0 \\ 3 \\ 2 \\ 2 \\ 1 \end{pmatrix} \qquad S = \begin{pmatrix} 0 & 1/6 & 1/3 & 0 & 0 & 0 \\ 1/2 & 1/6 & 1/3 & 0 & 0 & 0 \\ 1/2 & 1/6 & 0 & 0 & 0 & 0 \\ 0 & 1/6 & 0 & 0 & 1/2 & 1 \\ 0 & 1/6 & 1/3 & 1/2 & 0 & 0 \\ 1 & 1/6 & 0 & 1/2 & 1/2 & 0 \end{pmatrix}$$

Now that we have the probability to jump into a node $j$ from any node $i$, we can define the initial probability distribution $\mathbf{p}^{(0)}$ which depends on the starting node $j_0$.

$$p_i^{(0)} = \delta_{ij_0} \tag{3}$$

Let us introduce the Perron-Frobenius operator $G$:

$$G_{ij} = \alpha S_{ij} + (1 - \alpha)v_i \tag{4}$$

Where $\alpha$ is the damping factor and $\mathbf{v}$ a preferential vector. By convention, we take $\alpha = 0.85$, but it could be anything in the interval $[0.5, 1[$. The values of the preferential vector $\mathbf{v}$ are all the same: $v_i = 1/N$.

By applying $G$ onto $\mathbf{p}$ an infinite number of time (i.e. $G^{\infty}\mathbf{p}^{(0)}$) we find the steady state probability distribution $\mathbf{P}$.

The computation of the Google matrix in the example network gives:

$$G = \begin{pmatrix} 0.025 & 1/6 & 0.308 & 0.025 & 0.025 & 0.025 \\ 0.450 & 1/6 & 0.308 & 0.025 & 0.025 & 0.025 \\ 0.450 & 1/6 & 0.025 & 0.025 & 0.025 & 0.025 \\ 0.025 & 1/6 & 0.025 & 0.025 & 0.450 & 0.875 \\ 0.025 & 1/6 & 0.208 & 0.450 & 0.025 & 0.025 \\ 0.025 & 1/6 & 0.025 & 0.451 & 0.451 & 0.025 \end{pmatrix} \tag{5}$$

Once we have $\mathbf{P}$, we have to sort it in decreasing order and we have the ranking of our network. We now have all that is needed to solve the eigenproblem and rank the network.

After solving the eigenproblem, the results are as follows:

| Node | Rank |
|---|---|
| 4 | 1 |
| 6 | 2 |
| 5 | 3 |
| 2 | 4 |
| 3 | 5 |
| 1 | 6 |

Table 1: Ranking for Figure 1

| | Probability | | | | | |
|---|---|---|---|---|---|---|
| $\alpha$ | Node 1 | Node 2 | Node 3 | Node 4 | Node 5 | Node 6 |
| 0.50 | 0.11622 | 0.14530 | 0.12452 | 0.23893 | 0.17590 | 0.19910 |
| 0.55 | 0.10942 | 0.13953 | 0.11791 | 0.24966 | 0.17806 | 0.20539 |
| 0.60 | 0.10207 | 0.13270 | 0.11058 | 0.26159 | 0.18050 | 0.21253 |
| 0.65 | 0.09408 | 0.12468 | 0.10246 | 0.27480 | 0.18330 | 0.22065 |
| 0.70 | 0.08520 | 0.11503 | 0.09326 | 0.28979 | 0.18658 | 0.23012 |
| 0.75 | 0.07541 | 0.10372 | 0.08295 | 0.30665 | 0.19031 | 0.24093 |
| 0.80 | 0.06433 | 0.09008 | 0.07110 | 0.32611 | 0.19472 | 0.25364 |
| 0.85 | 0.05183 | 0.07390 | 0.05756 | 0.34846 | 0.19981 | 0.26840 |
| 0.90 | 0.03732 | 0.05415 | 0.04163 | 0.37486 | 0.20592 | 0.28608 |
| 0.95 | 0.02035 | 0.03005 | 0.02281 | 0.40623 | 0.21324 | 0.30728 |
| 0.99 | 0.00447 | 0.00671 | 0.00503 | 0.43599 | 0.22022 | 0.32754 |

Table 2: Results for different values of $\alpha$

Looking at Table 2, the action of the damping factor $\alpha$ on the calculation of the steady state probability is clear. As $\alpha$ increases, the values for the first 3 nodes decreases while the values of the last 3 nodes increases. By taking into consideration the damping factor in the computations, we prevent the program to stuck itself on a the last 3 nodes, therefore producing more accurate results.

# 3 Task 2

The approach of the adjacency matrix we saw in section 2 is satisfying for small networks. However, with a network or several thousand nodes, solving for the eigenvalues of a matrix is not a conceivable method, the computation time and the load on the memory would be to high. A new method is needed, the power iteration method.

## 3.1 About the program

In order to build a PageRank algorithm using the power iteration method, I choosed to use object oriented programming. My program is composed of multiple files:

- *main.py*: "Controller" of the program
- *parameters1.py*: Parameters to use for the task 1

- *parameters2.py*: Parameters to use for the task 2
- *function_task1.py*: Contains the **class NetworkTask1** that computes the rank using the adjacency matrix method
- *function_task2.py*: Contains the **class NetworkTask2** that computes the rank using the power iteration method

## 3.2 *main.py*

This file is made of 2 parts, depending on the value of **run_all_files** (bool) of *main.py*.

If **False**, it will run the task 1 and the task 2 on the small network (*network_data.txt*) and print the rank of each node for both tasks.

If **True**, it will run only the task 2 on a list of files (**files** in *parameters2.py*), printing several informations about the computation process and save logs containing time-stamp and the number of iterations needed for the convergence of P (*logs.log* is a file containing all the logs, the starting and ending time of the program, it also saves a log file for each computed network *"filename".log*).

## 3.3 *parameters2.py*

This file contains the constants to use for the task 2:

- The damping factor **alpha**
- The criterion of convergence precision **epsilon**
- The list of files to use **files**

## 3.4 functions_task2.py

This is the core of the program, where all the computations are made.

I start by defining a **class NetworkTask2** which is kind of a "storage" for all the values of the network.

Each function defined in the **class** measures and saves the execution time in a string **self.log**

There are some functions when I am forced to use lists instead of arrays (mostly because we cannot know the size of the array beforehand), I made sure to always return arrays, which are less expensive in memory.

I will not comment on the time-stamp lines in the pseudo-code since they are not relevant for the algorithm, they are here to keep track of the state of the program.

### 3.4.1 ___init___

This function is called when initiating the **class**, it is merely here to show what names are used for the different variables since I don't want to initiate them at **class** call.

### 3.4.2 compute

This function calls the other functions to compute each **attribute** of the **class**.

| Input | Output |
|---|---|
| • **filename (str)**: the name of the file to use | • **None** |

Table 3: **load_data** function variable

### 3.4.3 load_data

This function loads the data stored in **filename** and returns it in a array along with the number of nodes.

| Input | Output |
|---|---|
| • **filename (str)**: the name of the file to use | • **data (np.ndarray)**: the links between the nodes<br>• **n_node (int)**: the number of nodes in the network |

Table 4: **load_data** function variables

#### 3.4.4 build_degree

This function builds **k_out**

| Input | Output |
|---|---|
| **None** | • **k_out (dict)**: the number of ways to leave each node |

Table 5: **build_degree** function variables

It works by using the function **Counter()** of the package **collections** which counts the occurences for each value in the column 1 and stores them in a sort of **dictionary** (it is a specific type from the package **collections**, but easily converted to a **dictionary**) according to the following pattern:

```
{node1: occurence, node2: occurence, ..., nodeN: occurence}
```

#### 3.4.5 build_dangling

This function store the number associated to each dangling node (node without any way to leave).

| Input | Output |
|---|---|
| **None** | • **dangling (np.ndarray)**: the number of ways to leave each node |

Table 6: **build_degree** function variables

It makes use of the dictionary **k_out** and the number of nodes to build **k_out**. The function iterates on the number of nodes and checks if there is an entry for the current node. If there is none, it appends the current node to the list dangling.

#### 3.4.6 build_p

This function computes the steady state probability **p** of each node.

This function is the longest of the program, for it does several long tasks. It starts by initiating **gp** as an array the size of the network using the following definition : $Gp(i)_N = \frac{1}{N}$ with $N$ the number of nodes in the network. It then enters an infinite do while loop where it computes **p** until $||\mathbf{gp} - \mathbf{p}|| >$ **epsilon** or if the iterations limit (set at 1000) is reached.

Do While details:

1. Store a copy of **gp** in the variable **p**
2. Parse the list of links (**self.data**) and for each link A $\rightarrow$ B increase **gp**(B) by $\frac{\alpha * \mathbf{p}(\mathrm{A})}{\mathbf{k\_out}(\mathrm{A})}$
3. Parse the list of dangling nodes (**self.dangling**) and for each dangling node C increases **gp**(i) by $\frac{\alpha * \mathbf{p}(\mathrm{C})}{N}$
4. At each iteration, increase **gp**(i) by $\frac{1-\alpha}{N}$

| Input | Output |
|---|---|
| • **filename**: the name of the file to use | • **p (np.ndarray)**: the steady state probability of each node |

Table 7: **build_p** function variables

5. Divide **gp** by its norm.
6. Perform the checks, if the criterion of convergence is met or if the number of iterations is higher than 1000, the loop stops. Else it starts over with the newly computed **gp**.

After the loop, it returns the array **m**.

#### 3.4.7 build_index

Given the steady state probability **p** for each node, this function computes the rank of each node and sorts them.

| Input | Output |
|---|---|
| • **filename**: the name of the file to use | • **k (np.ndarray)**: each node and their rank |

Table 8: **build_index** function variables

This function initiates **k** as an empty list and appends a tuple (**node**, **p**) for each node. It then converts this list into an array and sorts it by probability (higher **p** first). Then it iterates on the number of nodes and assign a rank to each node (replace **p**(i) with **k**(i)).

About the means used in Python

In order for numpy to sort an array by a column, we must make use of a structured array. However, I could not simply create a structured array, so I built a list a tuples as said above. I then convert it to an array using the argument **dtype** to assign the columns to their name (**node** and **rank**). With that, I can tell numpy to sort the array by order of **rank**. Since it sorts from lower to higher, I flip the list to have the wanted order. After that, all that is left is to assign the ranks in order and save the array.

## 4 Performances comparison

Running both task on the 6-nodes network brings up the following results:

| | Task 1 | Task 2 |
|---|---|---|
| Computation time | 1.383 ms | 3.248 ms |

Table 9: Ranks and computing time for both tasks

It appears that the task 1 is faster than the task 2. It is probably because the network is to small, but the smallest sample of data we had (*thwiki.txt*) was already too big for the computation (it needs something like 45 GB of memory). Regarding the probabilities and ranks, here are the results:

The ranks are totally identical while the probabilities are only identical up to a certain degree of precision. It is set by $\epsilon = 10^{-4}$ which is the limit .

## 5 Task 3

Now that we have a full-functional algorithm to compute the PageRank, it is time to use it on real data. We have a list of files containing the hyperlinks between the Wikipedia pages of 24 languages. We are

| | Probability | | Rank | |
|---|---|---|---|---|
| Node | Task 1 | Task 2 | Task 1 | Task 2 |
| 1 | 0.05183647 | 0.05170476 | 6 | 6 |
| 2 | 0.07390771 | 0.07367929 | 4 | 4 |
| 3 | 0.05756534 | 0.05741243 | 5 | 5 |
| 4 | 0.34846758 | 0.34870366 | 1 | 1 |
| 5 | 0.19981617 | 0.19990381 | 3 | 3 |
| 6 | 0.26840673 | 0.26859606 | 2 | 2 |

Table 10: Probabilities and ranks for both tasks

interested in the first 10 pages of each language, a comparison of the rank of the same page in different language and the evolution of the CPU time needed in function of the number of links.

### 5.1 Top 10 articles

| Article | Rank |
|---|---|
| (310) | 1 |
| (61633) | 2 |
| (129120) | 3 |
| (250) | 4 |
| (637) | 5 |
| (779) | 6 |
| (571) | 7 |
| (1493) | 8 |
| (5996) | 9 |
| (35) | 10 |

Table 11: Top 10 articles for the Arabic edition

## 6 Conclusion