# Numerical Analysis for ML project: PageRank in Python

Matteo Negro

Course: NAML

Politecnico of Milan

March 12, 2023

POLITECNICO
MILANO 1863

# Contents

**Abstract**

This work will present an implementation of the PageRank algorithm introduced by Larry Page and Sergey Brin. I will also present an implementation of Graph ADTs in order to make the algorithm work properly. The aims of the project are about understanding the usage of objective-oriented programming in Python and learning how web surfers work. Browsers, such as Google, are used to rank nodes in the network (Graph) respect their importance. But how to do that? That is what *PageRanking* longs for. The paper [1] reports the methods developed by Larry Page and Sergey Brin. As an optional challenge the assignment asked to improve the performance in order to scale also on the bigger dataset. The reason why the algorithm needs to be efficient is almost self-explanatory. Think about Google and how many websites there are in our days. The code submitted consisted of 2 Python files. The first one with data structures and the second one with the implementation of the *PageRank algorithm.*

## 1 Introduction

In 1998, Michigan alum Larry Page and his fellow Stanford PhD student Sergey Brin introduced PageRank, a novel web search ranking algorithm. *PageRank* was the foundation of what became known as the Google search engine. More generally, *PageRank* can approximate the “importance” of any given node in a graph structure.

Intuitively, a node in a graph will have a high *PageRank* if the *sum of the PageRanks of its backlinked nodes* is high. Thus, a node’s *PageRank* depends not only on the number of backlinks it has but also on the importance of those backlinked nodes. For example, if a node $u$ only has one backlink from node $v$, but node $v$ has a high *PageRank*, then node $u$ will also have a relatively high *PageRank*.

A real-world example is the Twitter graph of who-follows-whom, where a directed edge from user A to user B exists if A follows B. Famous people like Barack Obama have millions of followers, some of whom are also famous with millions of followers. Thus, when represented as nodes in a graph, these users will have a high *PageRank*. Furthermore, such celebrities’ high *PageRank* will contribute to the *PageRank* of the users that *they* follow. If Barack

Obama follows another user and is that user's only follower, that user will still have a relatively high *PageRank*.

Note that this definition of *PageRank* is inherently recursive: computing a node's *PageRank* depends on other nodes' *PageRanks*, which in turn depend on other nodes' *PageRanks*, and so on. However, Page and Brin show that the *PageRank* algorithm may be computed iteratively until convergence, starting with any set of assigned ranks to nodes.

## 2 Methods

### 2.1 Theory behind algorithms

In this section, I will present the formula used to update the *PageRank* score of nodes. As I told you before, this formula will be computed till convergence following this update rule:

$$PR_{k+1}(u) = \frac{1-d}{N} + d\left(\sum_{v \in BL(u)} \frac{PR_k(v)}{v^+} + \sum_{\text{w is sink}} \frac{PR_k(w)}{N}\right) \tag{1}$$

In order to try to explain the formula above, we can just look at it piece by piece. The main idea of the algorithm is to give importance to nodes based on, again, the importance of the nodes linked to it. To model such behavior we take into account an average value of the old page ranking score of the *backlinked* nodes:

$$\sum_{v \in BL(u)} \frac{PR_k(v)}{v^+} \tag{2}$$

Following that reasoning, the sink nodes, the ones without outgoing edges, will not take part in the game. So what Page decided is to consider adding their contributions to each node. That is because when we explore the graph and we end up in a sink node, we have to restart from a random node. Assumed that we have to add this formula (3) to (2):

$$\sum_{\text{w is sink}} \frac{PR_k(w)}{N} \tag{3}$$

Finally, we introduce a *damping factor* $d \in (0, 1]$. This parameter models the fact that, actually, every time we can restart from a new node even if we are on a page that does have out-links. Thus, we damp a node's *PageRank* by multiplying it by $d$, and then we assume that the total residual probability 1 - $d$ is distributed among all nodes so that each page receives $(1$ - $d)/N$ as its share. Putting all this stuff together, we end up with the first formula (1).

## 2.2 Graph ADTs

The first file named *graph.py* presents the implementation of different ADTs. You can find a good implementation of the code following the Objective-Oriented paradigm. In the following sections, I present each class and its methods.

### 2.2.1 GraphError

This is a class for handling exceptions. It is used by the other ADTs. Its methods are briefly explained below:

***__init__*** **:** The constructor takes an optional argument representing the message associated with the exception, defaulted to an empty string.

***__str__*** **:** This method produces the saved message.

***__repr__*** **:** This method produces a code representation of the object.

### 2.2.2 Node

This class, instead, represents a node in a graph, with an ID and attributes. Here we can see its methods:

***__init__*** **:** The constructor initializes the node with the given ID and sets the optional attributes.

***__str__*** **:** This method produces a string that contains all the information about the node in a well-formatted way. Its attributes are printed sorted, increasing, lexicographic order.

***identifier*** **:** This method returns the identifier of this node.

***attributes* :** This method returns a copy of the node's attributes through a dictionary.

### 2.2.3 Edge

This class represents an edge in a graph between two nodes. Also here, an edge can have its own attributes and the following methods give access to them:

**__*init*__ :** The constructor initializes the edge with the two Nodes, node1 and node2. It also set the optional attributes of the edge.

**__*str*__ :** This method produces a representation of this edge and its attributes in sorted, increasing, lexicographic order.

***nodes* :** This method returns a tuple of the two Nodes corresponding to this edge. The nodes are in the same order as passed to the constructor.

***attributes* :** This method returns a copy of the edge's attributes through a dictionary.

### 2.2.4 BaseGraph

This is the main base class for both directed and undirected graphs. All the features of a graph are implemented here with the following methods:

**__*init*__ :** The constructor has no parameters, but when it is called creates several private data structures. Those data structures make manageable all the features that we can expect from a Graph, such as a counter of nodes, a list for the nodes themselves, a dictionary for the edges, and so on. This function has a complexity of *O(1)*.

**__*str*__ :** This method returns a string representation contains the nodes in sorted, increasing order, followed by the edges in order. This function has a complexity of $O(N \log N + E \log E)$.

**__*len*__ :** This method returns the number of nodes in the graph. This function has a complexity of *O(1)*.

***__getitem__*** **:** This method returns the Node or Edge corresponding to the given key. If the key is a node ID, it returns the Node object whose ID is key. Instead, if the key is a pair of node IDs, it returns the Edge object corresponding to the edge between the two nodes. This function has a complexity of *O(1)*.

***__contains__*** **:** This method returns whether the given node or edge is in the graph. If the item is a node ID, return True if there is a node in this graph with an ID item. If the item is a pair of node IDs returns True if there is an edge corresponding to the two nodes. Otherwise, returns False. This function has a complexity of *O(1)*, thanks to the fact that nodes and edges are stored in the dictionary.

***add_node*** **:** This method adds a node to the graph by calling the Node's constructor. This function has a complexity of *O(1)*.

***nodes*** **:** This method returns a list of all the Nodes in this graph.

***add_edge*** **:** This method adds an edge between the nodes with the given IDs. This function has a complexity of *O(1)*.

***edge*** **:** This method returns an Edge object for the edge between the given nodes. This function has a complexity of *O(1)*.

***edges*** **:** This method returns a list of all the edges in this graph. This function has a complexity of *O($E\log E$ )*.

***get_bl*** **:** This method returns a data structure that contains per each node $n$ a mask, where the True value is set if i-node is backlinked to n-node, with $i \in$ Set of Nodes.

### 2.2.5 UndirectedGraph

This class models the undirected graphs, extending the BaseGraph one. This ADT adds just a method and overrides another one, as follows:

***add_edge*** **:** This method returns the degree of the node with the given ID.

***degree*** **:** This method adds two edges between the nodes with the given IDs (Both directions).

### 2.2.6 DirectedGraph

Instead, this class represents a directed graph, derived from BaseGraph. In this class, I add some more methods in order to have more features, according to the requirements:

> ***__init__*** **:** The constructor simply calls the BaseGraph's one and only thing that adds a data structure to collect out-degree in a smart way.
>
> ***add_node*** **:** This method does the same as the BaseGraph. The new thing here is the initialization in the data structure names before.
>
> ***add_edge*** **:** This method does the same as the BaseGraph. The new thing is the increments of the out-degree.
>
> ***in_degree*** **:** This method returns the in-degree of the node with the given ID.
>
> ***out_degree*** **:** This method returns the out-degree of the node with the given ID.
>
> ***out_degree_vector*** **:** This method returns the out-degree of all the nodes.

## 2.3 Algorithm

The second file, named *pagerank.py*, contains the algorithms its-self.
In order to make it as clear as possible and also as fast as I can, I divided the function *pagerank* into macro areas. This organization helps me also to benchmark the function and find bottlenecks.

- In this section, I dump the nodes from the graph using the specific method of the class Graph.
- After that, I collect, in a dictionary, per each node $n$ the mask which contains the backlinked nodes of node $n$.
- Then, I compute the out-degree of all the nodes, storing that result in a vector.
- After that, I store a mask that tells me which ones are the sink nodes.

- In this section I compute the actual algorithm. As we can notice, there is a for loop for managing the number of iterations, required from the algorithm. We can also find the storage of the value because *PageRanking* required an *out-of-place* computation.

- In this last section, I prepare the output in the 'well-formatted' way required by the external printing function.

## 3 Performance

The paper provides me with three different datasets to test the correctness and the performance of the algorithm.

As an optional challenge, it is required to complete the execution under some time constraints. In order to be able to do that, I have to use 'good' programming practices and define new methods in the ADTs.

My implementation is able to scale and manage even the biggest of the given dataset. The *pageranking* algorithm has two hyper-parameters: the *number of iteration* and the *damping factor*. In the following presented data, these parameters are set up both to their default value: 40 and 0.85, respectively.

In the following section, I'm going to present the results of these runs, showing the dataset, time executions of each phase, and observation, where required.

## 3.1 Characters - Small Dataset

This is a toy example just to test the correctness of the algorithm. The graph represents the possible correlation among the characters of the famous cartoon Spongebob [Figure: 1].
The data are given in two *.csv* files: one for the nodes and one with the edges.

In this example, there are just 9 nodes and 8 edges. The following graph will not show the attributes of nodes and edges, for sake of simplicity.
The execution requires about 3.6 ms to compute and that is the result:

| NODE | $PR_{40}(n)$ |
|---|---|
| Mr. Krabs | 0.21345 |
| Spongebob | 0.13100 |
| Mrs. Puff | 0.09308 |
| Squidward Tentacles | 0.09308 |
| Sandy Cheeks | 0.09308 |
| Patrick Star | 0.09308 |
| Larry the Lobster | 0.07081 |
| Pearl | 0.07081 |
| Plankton | 0.07081 |
| Gary | 0.07081 |
| **Sum** | **1.00000** |

Table 1: Results of the *pageranking* alg.

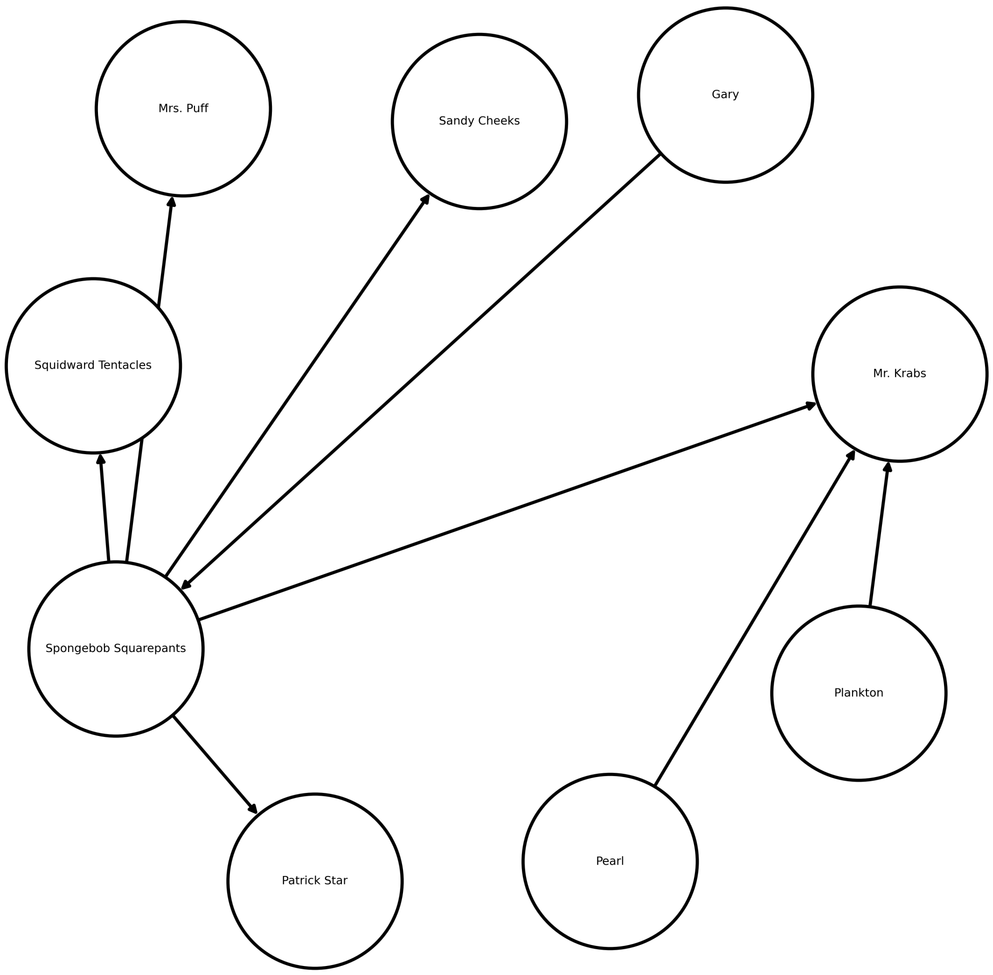

Figure 1: Graph representation of the small dataset.

In the following table [Table: 2] I show the time execution of each phase of the algorithm. As we can expect the 'stressful' part, computational speaking, is the execution of the algorithm due to the dump of old values and the for loop that encapsulates all this part.

| PHASE | TIME (ms) |
|---|---|
| Getting Nodes | 0.00600 |
| Eval. BackLinks | 0.02300 |
| Eval. OutDegree | 0.00900 |
| Eval. SinkMask | 0.01000 |
| Comp. Algorithm | 3.28900 |
| Prep. Printing | 0.02100 |
| **TOTAL:** | **3.35800** |

Table 2: Time phases of the algorithm.

## 3.2 Email - Medium Dataset

This dataset contains email users and binary links between them. If user 1 sends an email to user 2.

In this case, the number of nodes and edges is much bigger. We have 1006 nodes and 25572 edges. The time requires is about 0.3s. In Figure 2 I plot just 100 edges and the respective nodes, in order to give an idea of the complexity of the problem.

In the following table [Table: 3] I present the $9^{th}$ best nodes that the algorithm returns:

| NODE | $PR_{40}(n)$ |
|---|---|
| 1 | 0.00997 |
| 130 | 0.00729 |
| 160 | 0.00674 |
| 62 | 0.00531 |
| 86 | 0.00511 |
| 107 | 0.00499 |
| 365 | 0.00477 |
| 121 | 0.00471 |
| 5 | 0.00451 |
| ... | ... |
| **Sum** | **1.00000** |

Table 3: Results of the *pageranking* alg.

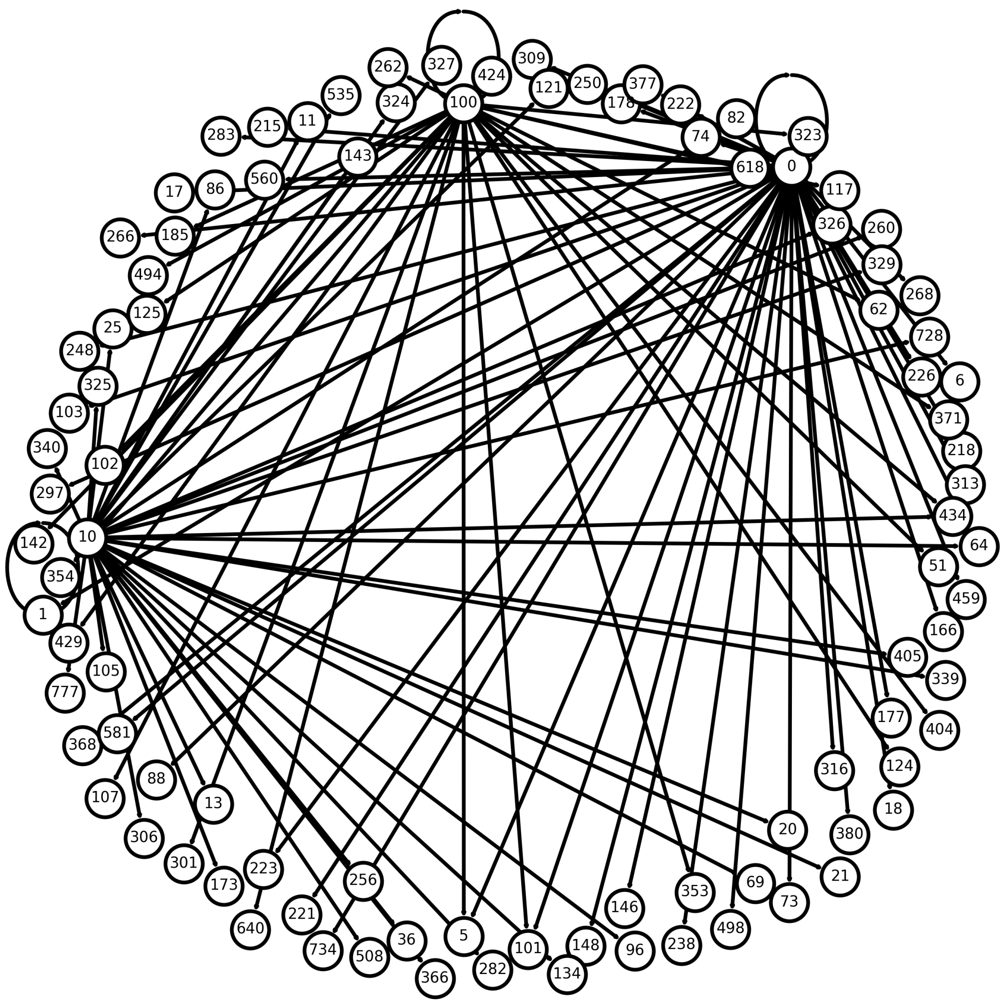

Figure 2: Graph representation of the partial medium dataset.

Also for this section, I present the time execution for each phase [Table: 4]. We can notice that the trend it's almost the same. Although the number of edges increases exponentially the time to compute the algorithm doesn't. The reason is that actually most of the operation can be done before the start of the for loop and that makes the algorithm efficient.

| PHASE | TIME (ms) |
|---|---|
| Getting Nodes | 0.13200 |
| Eval. BackLinks | 3.73700 |
| Eval. OutDegree | 0.23400 |
| Eval. SinkMask | 0.01600 |
| Comp. Algorithm | 312.14600 |
| Prep. Printing | 0.25800 |
| **TOTAL:** | **316.523** |

Table 4: Time phases of the algorithm.

## 3.3 Twitter - Large Dataset

The last dataset present is the bigger one, it is taken from the social network Twitter. The number of nodes and edges becomes huge. In particular, we have 81307 nodes and 1768150 edges.
In this case, being 'wise' and 'smart' is of paramount importance. In order to compute the algorithm of this dataset, I had to work not only on the algorithm its-self but also on the ADTs. Thus, I was able to reduce the times that I have to go through the edges. As for the previous case, I present first the score of the $9^{th}$ best nodes [Table: 5] and, again, a part of the dataset (250 edges) in order to give an idea of the problem's nature [Figure: 3].

| NODE | $PR_{40}(n)$ |
|---|---|
| 115485051 | 0.00894 |
| 116485573 | 0.00761 |
| 813286 | 0.00235 |
| 11348282 | 0.00133 |
| 40981798 | 0.00130 |
| 7861312 | 0.00120 |
| 15439395 | 0.00109 |
| 17093617 | 0.00105 |
| 15485441 | 0.00102 |
| ... | ... |
| **Sum** | **1.00000** |

Table 5: Results of the pageranking alg.

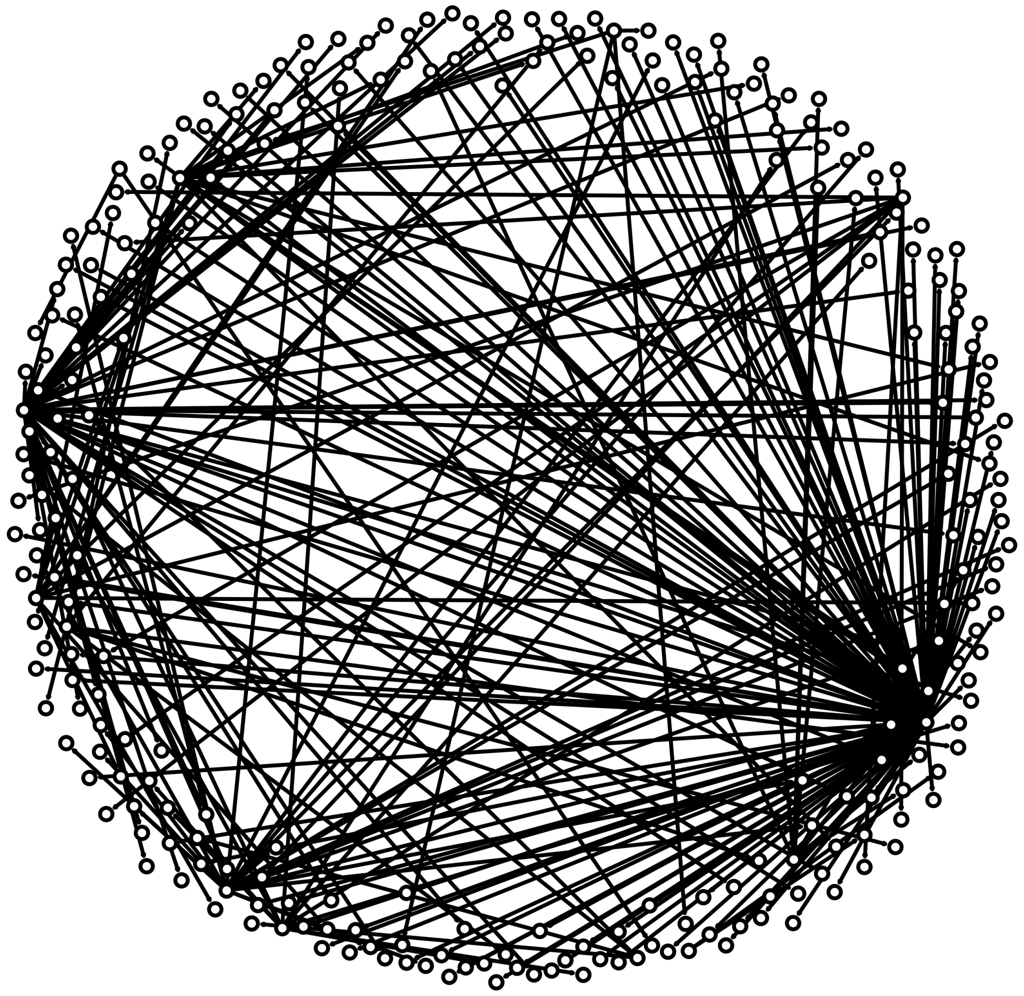

Figure 3: Graph representation of the partial big dataset.

As I said before, this dataset is actually huge. The overall time execution is around 2'30" on average.

In order to make this dataset manageable in a feasible time, I had to work on the ADTs. I decided to add some data structure in the graph to store in the building phase information about out-degree and backlinked. Thus, I do not need to go through more times edges. Actually doing that in the building phase is very cheap computationally speaking. The improvement in the performance was of more than *10x*.

| PHASE | TIME (s) |
|---|---|
| Getting Nodes | 0.01462 |
| Eval. BackLinks | 2.08120 |
| Eval. OutDegree | 0.04510 |
| Eval. SinkMask | 0.00016 |
| Comp. Algorithm | 157.77738 |
| Prep. Printing | 0.02723 |
| **TOTAL:** | **159.94569** |

Table 6: Time phases of the algorithm.

## 4 Conclusion

The paper aims to make you more familiar with the O.O. paradigms in Python and at the same time give you an idea of *PageRanking* algorithm. This algorithm is at the base of search engines, like Google. It is able to capture the real importance of nodes in a graph. Just weighting also the importance of the backlinked nodes. That is the crucial part, that is what makes this algorithm of paramount importance. My implementation as shown during the presentation is actually good and fast. It is able to manage huge datasets.

## References

[1] Project 1: PageRank in Python.