|  |
| --- |
| PageRank algorithm implementation project report  FOR THE NUMERICAL ANALYSIS FOR MACHINE LEARNIG COURSE (A.A. 2022/2023)  Author: **Matteo Pisani** (Codice persona: 10921568; numero matricola: 221978) |
|  |
|  |

# Abstract

The PageRank algorithm was developed by Google to evaluate the importance of a web page with respect to the World Wide Web link structure. The algorithm is based on a solid mathematical model, which can be defined by means of matrix algebra. This report presents a Python implementation of the PageRank algorithm which closely follows this algebraic formulation of the problem. The results have been shown to be coherent with respect to those obtained by standard implementations.

# Contents

[Abstract 2](#_Toc145968950)

[Contents 3](#_Toc145968951)

[1 Background knowledge: a mathematical model for PageRank 5](#_Toc145968952)

[1.1. Introduction to PageRank and random surfer 5](#_Toc145968953)

[1.2. The PageRank problem 5](#_Toc145968954)

[1.3. P-matrix constructions 6](#_Toc145968955)

[2 Project specification 7](#_Toc145968956)

[3 Solution design and implementation 8](#_Toc145968957)

[3.1. Overview 8](#_Toc145968958)

[3.2. Dataset 9](#_Toc145968959)

[3.3. PageRankCalculator: detailed implementation 10](#_Toc145968960)

[4 Testing and results 13](#_Toc145968961)

[4.1. Accuracy testing and results 14](#_Toc145968962)

[4.2. Performance testing and results 17](#_Toc145968963)

[5 Future developments and conclusions 20](#_Toc145968964)

[Appendix A 21](#_Toc145968965)

[References 22](#_Toc145968966)

*This page is intentionally left blank*

# Background knowledge: a mathematical model for PageRank

## Introduction to PageRank and random surfer

The PageRank algorithm was developed by Google in order to evaluate the importance of the pages resulting from a web query. Each page is assigned a value which depends on the number of its backlinked pages and, recursively, on their PageRank value; in this way it is possible to order the results based on their relevance and show only the most prominent ones to the user. The aim of the PageRank algorithm is thus to calculate this value for each page.

Before showing a more formal definition of the problem, we must introduce the concept of random web surfer [1]. This is an hypothetical user of the web who behaves in a specific way: when visiting a page, the random surfer can follow one of the outbound links available on the page or can “teleport” to a random page, even if it’s not an adjacent one. We call the probability of following the outlinks *alpha*, and the probability distribution over the pages when teleporting on *v*. The PageRank value of a given page can be interpreted as the stationary probability of visiting that page according to the random surfer model.

We could use random surfer models on a graph, therefore working with nodes and edges instead of pages and links. This allows us to formulate the problem in a more rigorous way, as shown in the next Section. We will assume to work on directed, non-weighted graphs.

## The PageRank problem

Let us now provide a mathematical representation of the problem using matrix algebra [1, 2, 3, 4]. Note that other formulations exist, but we will refer to the following one.

The structure of a graph can be described by a matrix **P.** Each element *Pij* represents the probability of reaching node *i* from node *j*, and it is equal to 1 over the out-degree of *j* if *i* is adjacent to *j*, otherwise it is 0. We will always assume the probability of visiting an adjacent node is uniformly distributed over all the nodes. **P** is a column-stochastic matrix, meaning that the sum of the elements over each column is equal to 1. Let us call **v** the vector containing for each node the probability of reaching that node in the case the surfer teleports; it is still a stochastic vector. Different values for **v** represent different teleportation behaviors: it could be a uniform vector, or it could focus only over a region of the graph; the assigned value depends on the problem we want to solve. **v** is also called personalization vector. Then, let us call *α* (or damping factor) the probability of the surfer to move on to an adjacent node during a certain time step, while 1-*α* is the probability of teleporting. The resulting PageRank values we will obtain depend on both **v** and *α*, which are part of the problem definition.

Given the previous concepts, we could define the PageRank problem as:

**Definition 1.1** *[2,4]: Let* ***P*** *be a column stochastic matrix,* ***v*** *a column stochastic vector and let 0<α<1. The PageRank problem is to find the solution* ***x*** *(the PageRank vector) of the linear system (****I*** *– α****P****)****x*** *=**(1-α)****v****.*

The PageRank vector contains the PageRank values for each node of the graph. As the matrix (**I** – α**P**) is a diagonally dominant matrix, the solution always exists and it is unique and non-negative.

The problem of computing **x** can be also solved using an iterative method, by applying the update rule:

(1.2)

It can be shown that this iteration has excellent convergence properties, which allow to find the solution in a finite and relatively small amount of epochs [2].

## P-matrix constructions

Let us now consider how we can compute the matrix **P** [3]. Let us take the adjacency matrix **A**, where each element *Aij* is equal to 1 if there is an edge from node *i* to node *j* and is equal to 0 otherwise. Let us call **d** the vector of node out-degrees for each node. Now we could compute each element of **P** like *Pij = Aji / dj*. However, we have to consider the case where some node has no outlinks, and therefore *dj* = 0; these nodes are called *dangling* (or *sink*) *nodes*. We can reformulate the computation of **P** as it follows:

(1.3)

We still have an issue with dangling nodes: as they have no node they link to, their associated column inside the **P** matrix is made of all zeroes, which makes **P** a sub-stochastic matrix, thus invalidating the definition we gave in the Section above and preventing us from finding a solution to the PageRank problem. So we must manipulate the **P** matrix found with the formula above so to make it column-stochastic.

For that purpose, there exists a variety of approaches: adopting one of those results in a different version of the algorithm. We will just take in consideration two of all the existing versions. The first one is called strongly-preferential PageRank. Here we assume that the surfer, once in a dangling node, will move to a node which follows the distribution of **v**. In practice, we update the matrix obtained with the formula (1.3) (which we will call ) in the following way. Consider **c** to be a vector so that *ci* = 1 if and only if *i* is a dangling node, and it is 0 otherwise. Then we apply the equation:

The second version is called weakly-preferential PageRank, where we assume that when reaching a sink node the surfer proceeds by transitioning to a random node following a uniform distribution; so we apply the rule

where **u** is a vector long as the number of nodes *N* and with value 1/*N* for each element.

# Project specification

After describing the theoretical background of PageRank, we can define the aim of the project in a more precise way. The objective was that of implementing the basic version of the algorithm (the one presented in the previous Section) and testing it over a set of simple graphs. The remaining design choices related to the algorithm were left unspecified and they were to be decided by the student: in particular, which version (or versions) of the algorithm to develop, which model of the problem to refer to, and which between exact and iterative approaches pick. We also had to design the module structure and the relationship among modules, the functionalities to implement in order to reach our main objective, and the way the interaction with the end user was to be. Finally, it was needed to define the correct testing strategies to assess the quality of the obtained results.

# Solution design and implementation

## Overview

The project was developed using the Python 3.10.0 language; it is structured in a total of 5 modules contained in the *src* folder of the project. The first one, *PageRankCalculator.py*, is conceptually the most relevant one, as it contains the implementation of the PageRank algorithm in the function *pageRank*. I chose to use the algebraic model of the problem given in Section 2 as my reference, therefore also in the code I make use of the same mathematical objects and analogous matrix operations. In that way the solution is elegant and concise, with also good performance, as it will be discussed later on. For what concerns the version of the algorithm, I chose to implement the weakly-preferential one. I decided to implement both an exact version of the PageRank, which solves the linear system of the problem directly, and an iterative one, which makes use of the iteration described in the previous Section, even though with some implementational differences. Both versions receive the damping factor *α* and the graph they work on as external input; the personalization vector is considered to be a uniformly distributed one. Finally, the module also contains a function which is responsible for computing the **P** matrix starting from a graph.

The *Graph.py* module, as the name suggests, contains the implementation of the Graph and the Node classes, which are used to represent the graphs which will be given as input to the *pageRank* function. As a design choice, I decided to focus on the graphs and nodes aspects which are more relevant in the context of the PageRank algorithm: therefore the definition of the classes is limited to the methods and fields which I found useful for the problem, and thus they are reduced to the essential. In this way I had the opportunity to focus more on the most relevant aspects of the project, which were those related to the algorithm *per se*. The two classes were much inspired by the work of Tony Chou [5], which code was already optimized for the PageRank problem and thoroughly tested. In particular, Graph objects allow to find nodes, add new nodes, add an edge between two nodes and get the adjacency matrix of the graph. Nodes only have one property, an identifying name, and edges have no weight or associated property.

The *GraphConstructor.py* module contains utility functions for generating graphs. In particular, the procedure *build\_graph* creates a Graph instance starting from a text file, where the graph is represented using the edge list convention. The function *generate\_edge\_list*  on the other hand generates a valid text file which encodes a randomly generated graph. One thing to note is that the classes from the Graph module are mostly used within these two functions, so only when generating a graph and not when executing the PageRank algorithm. That is why I’m not going to dive into the implementation details the *Graph* and *GraphConstructor* modules, as they are not as relevant as the *PageRankCalculator* one.

There is then the module which allows the user to execute the algorithm over a given graph; the module’s name is simply *pagerank.py*. It basically consists of a parser which extracts the file name, the damping factor and other parameter values from the user’s prompt. For a detailed description of how to invoke the module through the command line, see Appendix A.

The module *test.py* contains the test cases for the PageRank algorithm implementation. The strategy used for assessing the results is the following: the values obtained with my implementation are compared to those obtained using a trustworthy library implementation of the same algorithm. In particular, I chose NetworkX’s PageRank as a reference for testing. The results are deemed acceptable if and only if they differ from NetworkX’s by a small fraction of the real value. To test also the performance of my implementation, I also compared the average execution time of the *pageRank* function with that of NetworkX analogous procedure, using the *timeit* Python module. The execution of tests over a certain graph can be invoked by command line similarly to *pagerank.py*; see Appendix A for the details.

To implement the different modules, I made intensive use of two Python libraries: *numpy*, which was used wherever it was needed to perform matrix operations (for example in *PageRankCalculator.py*); NetworkX and *timeit*, as previously mentioned, for testing purposes. *numpy* was also used for testing, in particular its sub-library *numpy.testing* for assertions. It must be noted that, to avoid conflicts between libraries, a NetworkX version lower or equal than 2.6 is needed.

## Dataset

The dataset used to test the quality of the implementation consists of a set of thirteen text files (named *graph\_#.txt*, where # is comprised between 1 and 13), each containing a graph encoded using the edge-list convention, which are contained in the *dataset* folder of the project. As we anticipated, a file is given as input when invoking the execution of the PageRank or of the test code. The node size of the graphs ranges from 4 to 19500. The smaller ones were taken from Tony Chou’s repository [5], and they range from 4 to almost 500 nodes. The bigger ones were either auto-generated using the *GraphConstructor.generate\_edge\_list* or were found on Kaggle; those graphs range from a thousand to 19500 nodes. The bigger the graph, the more different and slower my implementation was with respect to the reference one; this point will be dealt with in the Testing and Results Section.

## PageRankCalculator: detailed implementation

Let us now consider *PageRankCalculator.py* in more detail. The module’s most relevant functions are: *P\_matrix*, *pageRank*, *pageRank\_exact*. This Section will describe their signature, the code-level details, the design choices that lead to their implementation and also will make some efficiency considerations. All the functions make use of the *numpy* library for working with vectors and matrices. As a leading design choice, the implementation will closely follow the theoretical framework presented in Section 2.

![Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente]()The *P\_matrix* function, as the name suggests, computes the **P** matrix (see Section 1) to use in the PageRank algorithm for a given graph. The complete signature is *P\_matrix(graph)*, where *graph* is an instance of the class Graph of the *Graph.py* module. It returns a 2 dimensional *numpy* array as a result. It is important to note that this method does not embed a policy for column completion in case of sink nodes: this means that it returns a sub-stochastic matrix. Instead, the completion policy is enforced inside the *pageRank* method, as it is a characteristic of the particular version of the algorithm.

*Figure 3.1: P\_matrix function code*

Let us analyse the code in Figure 3.1. Firstly, the adjacency matrix **A** is retrieved from the Graph object passed as input using the instance method *adjacency\_matrix*; it is also transposed. Then we compute the vector **d** of the out-degrees with *AT.sum(axis==0)*, which sums the values over each column. What the theoretical formulation would suggest now is to compute the **D** matrix starting from **d** and then computing its pseudoinverse. However this operation is very expensive, as it costs *O(N3)*, where *N* is the number of nodes in the graph. Instead, we perform the following equivalent operation: we divide each column of **AT** by the correspondent element of **d**, exploiting the broadcasting feature of the Python language. To avoid divisions by 0, we set each *di* = 0 to 1, which will return the correct result: each element of the columns associated with sink nodes will be divided by 1, thus it will remain equal to 0, which is the desired output. Thus we obtain the substochastic version of **P** and we return it to the caller; the asymptotic complexity of the procedure is reduced to *O(N2)*.

The *pageRank* function contains the implementation of the PageRank algorithm. Therefore, this method computes the PageRank value for each node of a graph, given some parameters which will be introduced soon. The procedure applies the weakly preferential policy for sink nodes, while the personalization vector is assumed to be always a uniformly distributed one. It also allows to choose between solving the PageRank linear system directly (exact solution), or applying an iterative method using the update rule described in Section 1; both approaches yield similar results in terms of accuracy, as it will be discussed in the Section dedicated to testing. Let us now consider the signature of the method as shown in Figure 3.2. The function receives the following parameters:

* *graph* (Graph): it is the Graph object containing the nodes to compute the PageRank for;
* *alpha* (float): the damping parameter that will be used in the execution. It is an optional parameter with default value of 0.85;
* *max\_iterations* (integer): the maximum number of iterations to do in case of iterative algorithm, after which the execution stops and the result is returned. By default it is equal to 400;
* *algo* (string): this parameter allows to select between the iterative and the exact methods for the problem resolution, by assigning the string “iterative” or “exact” to it. The default value is “iterative” in case this parameter is not provided a value by the invoker;
* *rround* (string): used to apply a rounding of the PageRank values to the first 3 decimal digits if the parameters equals “yes” (which is also the default value).

The function returns a *numpy* array containing the values for each node. The order of those values is that of the nodes inside the given *graph*, which are in turn stored following the order they appear in the text file they were generated from; this means the order of results does not follow any sorting criterion over node identifiers.

Consider the function’s code in Figure 3.2. In the case the invoker asked for the exact solution of the problem, *pageRank\_exact* is called, which contains the implementation of such approach; otherwise the execution of *pageRank* proceeds. We compute the P matrix by invoking the *P\_matrix­* function defined in the same module; remember that it is a substochastic matrix. We compute **c**, which is used to identify dangling nodes, as its elements are 1 if the associated node is dangling (i.e. the sum over its associated column in **P** is 0) and 0 otherwise. We also compute **v** that, as mentioned before, represents a uniformly random teleportation behaviour.Then the PageRank vector **x** is initialized in the same way of **v** using the *copy()* method; practical experience suggests that starting with this value for **x** results in a faster method with respect to initializing **x** as a zero vector [2]. The *dangling* variable represents the transition probabilities when reaching a sink node: as we adopted the weakly preferential version of the algorithm, it is uniform over all nodes as well as **x** and **v**. This vector will be useful when completing the **P** matrix, as it will be described in the following.

*![Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente]()Figure 3.2: pageRank function code*

We then have the convergence loop, which ends either when the error is less than a threshold comparable to the machine epsilon or when we reach the maximum allowed number of iterations defined by the user. In the loop’s body, we apply the update rule and we compute the error as the L1 norm of the difference between the new and the old **x** of two following iterations. Here there is a slight difference in the update of **x** with respect to our theoretical framework: we do not complete the **P** matrix, but instead we add another term which accounts for the sink nodes in the same way the completed **P** would do. Let us prove this important result. When multiplying the substochastic **P** for **x**, we obtain a vector **a** where each value *ai* is equal to the sum of all nodes’ PageRank values (contained in **x**) weighted for their likelihood of getting to node *i* (i-th row of **P**). Therefore the PageRank values of sink nodes do not appear in the summation, as the probability of getting anywhere from such nodes is null. Following the weakly preferential version of the algorithm, we should have the sink nodes values in the summation, multiplied for 1/*N*, where *N* is the number of nodes in the graph. To obtain this result, we add the term *np.inner(c, x\_old) \* dangling* to the product between **P** and **x\_old**. By computing the product between **c** and **x\_old**, we get the sum of the PageRank values for all sink nodes; we then multiply this result for the *dangling* vector, which is equal to 1/*N* for each element, thus obtaining the “discounted” PageRank contributions from sink nodes. This design choice was made as the cost of the operation described above is linear with respect to the number of nodes of the graph, while computing explicitly the **P** matrix is quadratic; this suggestion is inspired from Armin Sajadi *Fast Page Rank* implementation [6]. Lastly, we check the *rround* parameter and we round the result if requested by the caller.

![Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente]()The last function of the module is *pageRank\_exact*, which provides the same functionality of *pageRank*, but employing the exact method, which consists of solving the linear system described in the Subsection 1.2. It receives the same parameters of the iterative version (minus the *algo* one), which hold the same meaning and have the same default values. For what concerns the code in Figure 3.3, the first lines are comparable to the ones in *pageRank*. However, differently from the previous function, we do compute explicitly the complete **P** matrix, as it is necessary to define it directly to solve the linear system. To do so, we follow the definition of **P** in the weakly preferential version of the algorithm: after computing **c** (same vector as the standard *pageRank* function) and a uniform vector **u**, we perform the outer product and we add it to **P**. We then solve the system using the *linalg* library of numpy and return the result, after rounding it if requested.

*Figure 3.3: pageRank\_exact code*

# Testing and results

**­­**Let us now consider how the PageRank implementation has been tested and the obtained results. The general approach used was to compare the results given by the custom implementation with those of a standard PageRank library; as anticipated, it was chosen to use NetworkX implementation as a reference in the testing process. NetworkX employs by default the weakly-preferential version of the algorithm and a uniform initialization for the PageRank and personalization vectors [7]: so it applies the exact same version of the algorithm I have used and therefore the comparison is legitimate and meaningful.

The module consists of two test cases: the first one is used to check if the output values of my implementation are within a certain tolerance limit with respect to those of NetworkX (accuracy testing), for a given execution on a graph; the second one aims to compare the average execution time of the two implementations given a graph (performance testing). These will be explained in detail in the following subsections.

To execute the test cases on a given graph from command line, see Appendix A. It can be added that when invoking the *test.py* module over a graph, both the accuracy and the performance testing are executed, for each of the two custom implementations (iterative and exact); the results are printed on the screen.

## Accuracy testing and results

Accuracy testing is done by means of the *accuracy\_test* function. It assesses how closely the *pageRank* outputs match the ones produced by NetworkX’s analogue *pagerank* method; the custom algorithm will pass the test over a certain graph if the error made for each node is less than a tenth of the desired NetworkX value. Moreover it computes and prints the Mean Square Error (MSE) value between the two results and the number of nodes which hold the same placement when they are ordered by PageRank value. As parameters, it receives the *file\_name* of the graph and the damping parameter *alpha* to be given to the functions and an *algo* parameter to specify whether to use and test the iterative or the exact version of my implementation.

Let us now consider the *accuracy\_test* code in full detail. The function makes use of the support method *nx\_pagerank* to invoke NetworkX’s PageRank with a given graph and *alpha*: it creates a NetworkX *DiGraph* object, invokes the *pagerank* function and returns the results, which are stored in *nx\_result*. Its implementation is trivial and it won’t be discussed in this report. Then the custom *pageRank* procedure is called over the same graph using the same damping factor and the results are stored in the *my\_result* variable. For both the functions we do not perform any rounding, to better compare the similarity between results. We can observe that both functions return the PageRank values sorted by the same order, that is the one in which the nodes appear in the input file; thus it is easier to compute the MSE, by just doing the mean of the elements over the square-difference vector. Line 91 in Figure 4.1 counts the number of nodes that maintain the same placement when sorted based on their PageRank values in both implementations. This is achieved by comparing the sorted indices of the two output arrays using *np.argsort*. The result is stored in the *same\_order\_num* variable.

![Immagine che contiene testo, schermata, software

Descrizione generata automaticamente]()Then the results of the two implementations are compared in a more precise way, using tolerance levels. We first check if, for each node, the difference between its associated values in *my\_result* and­ *nx\_result* is less or equal to one thousandth (1/1000) of the *nx­\_result* value. If true, the test is successful and the relative error is displayed to the user; if false, a less strict check is performed, being whether the relative error is less or equal to one hundredth of the *nx­\_result*. Similarly as before, if true the execution passes the test, otherwise we check if the relative error is less or equal than a tenth: this is the last option and if it fails the whole test does. In the end, the MSE and the *same\_order\_num* variable are printed on the standard output. The relative error check is performed using the assertion method *assert\_allclose* of the *numpy.testing* library, which takes as input the actual and desired vectors and checks if any element violates the relative tolerance given by *rtol*. If this happens, an *AssertionError* exception is raised. Therefore, to implement the tolerance levels mechanism explained above, I used three nested try-except blocks: each one checks on a certain relative tolerance value and, if it fails, catches the exception and moves on to the next check, until there are no more. The captured exception is displayed to the user, showing that the previous check failed and some information about the error: the number of elements that don’t respect the relative tolerance set and the maximum absolute and relative difference between two elements of *my\_result* and *nx\_result*. These data are contained in the *AssertionError* object captured after a failure. An example of displayed results of a test run is shown in Figure 4.2.

*Figure 4.1: accuracy\_test code. The truncated part contains only the error message in case of failure of the assertion.*

![Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente]()*Figure 4.2: test execution console output*

Let us consider the accuracy test results over the dataset. We can generalize those results and group them with respect to the size of the graphs. For each group, we considered both sparse and dense graphs, obtaining comparable results despite their differences. All those results refer to the default *alpha* value (see the Appendix).

* For “small” graphs (here we mean graphs with less than 10 nodes and 100 edges), the iterative version of the custom implementation always obtains results which differ from the desired ones by less than 1/1000 fraction of the value. The MSE error is between e-15 and e-13, so close to the machine epsilon of the target machine (64 bit architecture), and the ordering of the nodes according to the custom PageRank is the exact same of that of NetworkX. For the exact version, the same results hold, with the exception of very small inaccuracies due to rounding error, which are unavoidable. This makes the ordering of nodes not exactly the same as the reference library; this phenomenon is more evident when the real PageRank values are very close to each other.
* For “medium-sized” graphs (meaning less than 500 nodes and 2000 edges), both the iterative and the exact version results differ from NetworkX’s by a hundredth of its value; the MSE is of the order of e-12. We can observe that, when working on the same graph, both version have the same relative error and the very close MSE; this consideration holds in general. For what concerns the ordering, we still have that the iterative version yields better results (the majority of the nodes have the same exact placement) than the exact one (less than half of the nodes have the same placement).
* For big graphs (less than 3000 nodes and 30000 edges), both versions of my PageRank result in a relative error of less than 1/10 and an MSE of the order of e-12 – e-11. The ordering of the nodes is the same as the reference library for a percentage which varies between 50% and 5%, for both versions. It can be seen that the more the graph is big, the more the accuracy of the custom implementation decreases, as it needs to deal with smaller PageRank values, which requires very high precision.
* For very big graphs (less than 20000 nodes and 200000 edges) we have results like the ones obtained for big graphs, even if the execution is considerably slower, as it will be discussed in the next Subsection. When the size of the graphs grows over the 20000 nodes threshold, the test usually fails, as the relative error grows up to 50% of the real value. Still, we have that the MSE and the ordering results are similar to the ones obtained for big graphs.

Overall we can say that the implementation contained in the *PageRankCalculator* module is precise enough when dealing with small-real world graphs and it could be used in contexts where this version of the algorithm is suited for.

## Performance testing and results

The function *time\_test* is dedicated to performance testing: its aim is to compare the custom PageRank average execution time with that of the standard implementation, over a given graph. To do that, the *pageRank* execution is repeated a number of times and the total time taken for the computation is averaged; the same is done for NetworkX’s function and the two results are compared. The standard Python module *timeit* allows to perform this operation in a transparent way and it is therefore used in the *time\_test* procedure. Similarly to accuracy testing, the test is meaningful as we use the same parameters and the same algorithm version for both my PageRank method and that of NetworkX. It should be noted that NetworkX’s implementation is an iterative one, while the custom one can be iterative or exact: therefore the comparison for the exact version will be slightly biased, as we are not considering exactly the same implementational approach. Both custom versions will be tested when invoking the *test.py* module, after the accuracy test.

The *time\_test* function takes as input *file\_name*, *alpha* and *algo*, which have the same meaning described in *accuracy\_test*. Moreover, it takes the parameter *number*, which represents the number of repeated executions used to measure average performance; as the execution time could be considerable for big graphs, *number*’s default value is 10. Finally, the last input is *nx\_time*, which is the reference NetworkX time the custom implementation should be compared to; if not given, it is computed inside the function. The default value is None, which is means that no reference time is provided.

![Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente]()Let us take in consideration the code in Figure 4.3. First we check if the *nx\_time* parameter is None: if true, we call the *timeit* function of the homonym module. The *timeit* method [9] receives as input a string containing the code to execute (in this case, an invocation to NetworkX’s PageRank implementation) and returns the overall amount of time of *number* executions of such code. To do so, the variables within the string need to be visible outside the *time\_test* method, so they’re added to the global scope at lines 152-155. We then apply *timeit.timeit* to our custom implementation, passing the same parameters used before. The two results are later compared to check which implementation was the fastest. In the end, the average custom function execution time is computed and displayed, together with the difference with respect to the average execution time of the standard implementation. The method returns the two averages: this is done so that successive invocations of the method over the same graph don’t need to invoke the *timeit* function for NetworkX’s PageRank again, as the second returned value of the previous invocation can be passed as the *nx\_time* input parameter; this allows to save some time.

*Figure 4.3: time\_test function code*

For what concerns the results, it is important to observe the following. The average PageRank implementation time over a certain graph is actually a random variable: it is the sample mean of the times for a finite number of PageRank executions. For each test run the PageRank executions may take different times following an unknown probability distribution: therefore the averages vary between different test invocations. In general, the bigger is *number* the more the sampled average is close to the real one; this is contrast with the feasibility of the test case execution, as calling *timeit* with a high *number* value is very expensive for big graphs. In conclusion, the obtained results are not deterministic ones, but they vary with each test run; however, we can observe they respect the following trend.

* For small graph (see the Subsection 4.1 for the number of nodes and edges) the average execution time is between the order of 10-4 and 10-3 for both iterative and exact versions; the exact algorithms runs tend to have slightly better performances. Usually it occurs that the custom implementations are faster than the library’s.
* For medium sized graphs, the average execution time grows and it is of the order of 10-2, getting much closer to 10-1; in particular, the performance of the exact version of my PageRank starts deteriorating, as it is no longer better than the iterative one. The reference performance is still around 10-3 seconds: so the custom implementation starts to gets worse than NetworkX’s.
* When dealing with big and very big graphs, it becomes evident that the average execution time depends not only on the number of nodes, but also on the number of edges. Indeed, for graphs with similar node size and very different edge size it occurs that, on average, the performance of the ones with more edges is considerably worse; while big graphs with many nodes but less edges have a performance that gets closer to that of medium-sized graphs. For instance, a graph with 1210 nodes and 1210 edges has an average execution time of less than one second; while a graph with 1005 nodes and 25572 edges of around 5 seconds. In general, the execution time is comprised between tenths of seconds for smaller graphs (in terms of both nodes and edges), while for huge graphs (meaning tens of thousands of nodes) it could take from some seconds (if the number of edges is less than 100000) to 3-4 minutes. Thus we can say that the algorithm tends to perform better over sparse graphs than on dense ones; this holds for both iterative and exact versions. In general, the bigger the graph, the more the exact version gets slower compared to the iterative one; this is something expected, as we know that solving a linear system exactly gets more computationally expensive the more variables we have, which are associated to the nodes. Thus the difference between iterative and exact versions is more and more evident as the number of nodes increases. Of course, the reference implementation for page rank does not suffer this problems and keeps having very good performances, under 1 second for any graph. This can be explained by the following. The reason why the computational cost of invoking *pageRank* grows so much with the number of nodes and (even more) edges is because of the computation of the stochastic matrix **P** depends linearly on the number of edges (as we first compute the adjacency matrix) and quadratically on the number of nodes. NetworkX’s uses special functions and encodings of the **P** matrix which make this procedure more efficient [7].

In conclusion, we can say that the custom implementation is not suited for working over relatively big graphs in a reasonable time in a real-life scenario, even if it computes fairly accurate values.

# Future developments and conclusions

In conclusion, this report described the implementation of a basic version of the PageRank algorithm, following the algebraic model of the problem. The obtained results have been proved correct by the comparison with a standard PageRank library implementation.

Therefore the objective defined in Section 2 has been achieved successfully in terms of correctness of the output. Still, the time performance of the algorithm is far from optimal and it could be improved in the future by using more efficient methods for matrix manipulation. Moreover new functionalities may be added to extend the scope of the application, for instance the possibility of specifying the personalization vector when applying the algorithm.

# Appendix A

The *pagerank.py* module can be invoked using the following sintax:

*python pagerank.py -f <graph\_file\_name>*

*[ --alpha <damping\_factor>*

*--round <”yes” if round results, “no” otherwise>*

*--iterations <the number of maximum iterations>*

*--algo <”exact” if use exact implementation, otherwise “iterative> ]*

The command receives as input one of the file names contained in the *dataset* folder (i.e. “*graph\_1.txt”*), between double quotes. It is possible to specify the following parameters: the damping parameter, which default value is 0.85; a parameter *round* indicating if the result has to be rounded or not; the number of maximum iterations in case of iterative version (default 400); finally, the *algo* parameter allows the user to specify which version of the algorithm they want to use (default “iterative”). The resulting values are displayed for each node on the standard output.

The *test.py* module can be invoked by:

*python test.py [ -f <list of graphs>*

*--alpha <damping factor>*

*--times <number of executions to average>]*

In this case we are allowed to provide more than one graph file as input: the test cases will be executed over each one of the given files and the results displayed. To do so it is necessary to write all the desired file names separated by commas and all within double quotes. By default, if we do not provide any file, the test cases will be executed over all files of the dataset until the “very big” graphs (meaning until the eleventh graph). For the damping parameter, the same rules for *pagerank.py* apply. The last parameter *times* is used to choose how many repeated executions of the algorithm will be performed to assess its average execution time; by default it equals 10.

# References

[1] Gleich, D.F. (2015). *PageRank Beyond the Web*. Society of Industrial and Applied Mathematics Review, Vol.57, No. 3, pp. 321-322

[2] Gleich, D.F. (2015). *PageRank Beyond the Web*. Society of Industrial and Applied Mathematics Review, Vol.57, No. 3, pp. 323-324

[3] Gleich, D.F. (2015). *PageRank Beyond the Web*. Society of Industrial and Applied Mathematics Review, Vol.57, No. 3, pp. 326-329

[4] Langville, Meyer (2006). *Google’s PageRank and Beyond. The science of search engine rankings.* Princeton University Press, pp. 71-75

[5] Chou T., *PageRank-HITS-SimRank*, <https://github.com/chonyy/PageRank-HITS-SimRank>

[6] Sajadi A., *Fast-PageRank*, <https://github.com/asajadi/fast-pagerank>

[7] NetworkX documentation, [https://networkx.org/documentation/stable/reference/algorithms/ generated/networkx.algorithms.link\_analysis.pagerank\_alg.pagerank.html](https://networkx.org/documentation/stable/reference/algorithms/%20generated/networkx.algorithms.link_analysis.pagerank_alg.pagerank.html)

[8] Numpy documentation, [https://numpy.org/doc/stable/reference/generated/numpy.testing. assert\_allclose.html](https://numpy.org/doc/stable/reference/generated/numpy.testing.%20assert_allclose.html)

[9] timeit documentation, <https://docs.python.org/3/library/timeit.html>