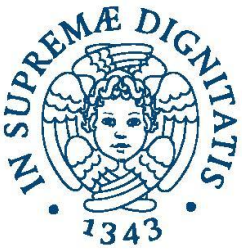
Pagerank application*

ANDREA PALAZZO†, University of Pisa, Italy



UNIVERSITÀ DI PISA

This paper provides a look to the design and implementation of the *Pagerank*, an algorithm used by Google Search to rank websites in their search engine results. There're different solutions related to this problem; in this paper we focus on an iterative approach, called *iterative method*, which computes the Pagerank cyclically using the previous value of Pagerank. In this paper we compute the Pagerank on a realistic dataset of web pages, the BerkStan. The solution has been implemented through *Hadoop*, a Java framework for distributed storage and processing of big data using the MapReduce programming model. In the next pages will be described the problems associated to Pagerank, the choices made in the implementation phase and the experimental results in Hadoop.

Additional Key Words and Phrases: search engine, MapReduce, distributed computation, Markov chain

ACM Reference Format:
Andrea Palazzo. 2018. Pagerank application. 1, 1, Article 1 (June 2018), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Application developed using Hadoop framework
†Student in Master degree Computer Science

Author's address: Andrea Palazzo, University of Pisa, Computer Science Department, 56100, Pisa, Italy, andrea.palazzo89@gmail.com.

XXXX-XXXX/2018/6-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

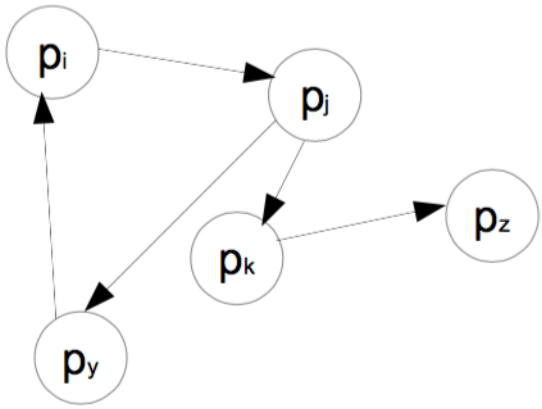


Fig. 1. Example of a webgraph

1 INTRODUCTION

The web network is expanding more and more, day after day, and so there are more and more users who use it. A lot of them use Google, or others search engines, to find one or more web pages that contains one or more keywords. For this reason we can imagine that a user is a web surfer that jumps from one page to another, following the hyperlinks contained in these pages.

For this reason we can view the web as a directed graph in which each web HTML page is a node and each hyperlink a directed edge. An example is in Fig.1, where page p_i has an hyperlink pointing to page p_j, page p_k points to page p_z, and so on. This graph is called **webgraph**. This graph is not strongly connected: there are pairs of pages such that one cannot proceed from one page of the pair to the other by following hyperlinks. We refer to the hyperlinks into a page as **in-links** and those out of a page as **out-links**; the number of in-links to a page is called **in-degree**, while the number of links out of a page is called **out-degree**.

The hyperlink from a generic page A to a generic page B represents an endorsement (or vote) of page B, by the creator of page A; the more votes are cast for a page, the more important the page will be. Moreover the anchor text pointing to page B is a good description of page B. However in this paper we focus on scoring and ranking measures derived from link structure alone. The most important technique for link analysis assigns to every node in the web graph a numerical score between 0 and 1: this technique is called **Pagerank**. Google recalculates Pagerank scores each time it crawls the Web and rebuilds its index.

We can view a user web as a *random surfer* who begins at a web page and executes a **random walk** on the web as follows: at each time step, the surfer proceeds from his current page A to a randomly chosen web page that A links to. The surfer proceeds at the next time step to one of the outgoing linked nodes, with probability 1/*out-degree*. As the surfer proceeds in this random walk from node to node, he visits some nodes more often than others; the idea behind Pagerank is that pages visited more often in this walk are more important.
It can happen that a page hasn't out-links, for this reason the random surfer can use the *teleport* operation. With this operation the surfer jumps from a node to any other node in the web graph,

even if there isn't any hyperlink between them; this could happen because he types an address into the URL bar of his browser. If N is the number of nodes in the web graph, the teleport operation takes the surfer to each node with probability *1/N*.
So the Pagerank score must take into account these two situations:

- when a node has no outgoing links, then the surfer uses teleport operation;
- when a node has outgoing links, then the surfer either follows one of outgoing links (with probability $0<d<1$) or invokes the teleport operation (with probability $1-d$).

The parameter d is called **damping factor**, chosen in advance, but its standard value is 0.85.

1.1 Problem statement

From the mathematical point of view we can see a random surfer on the web graph as a **Markov chain**[1], with one state for each web page, and each transition probability representing the probability of moving from one web page to another. The teleport operation contributes to these transition probabilities. A Markov chain is a process that occurs in a series of timesteps in each of them a random choice is made. A Markov chain is characterized by an N x N transition probability matrix P each of whose entries is in the interval [0, 1], where N is the number of web pages. The Markov chain can be in one of the N states at any given timestep; the entry P_{ij} tells us the probability that the state at the next timestep is j, conditioned on the current state being i. We can depict the probability distribution of the surfer's position at any time by a probability N-dimensional vector x. At t = 0 the surfer may begin at a state whose corresponding entry in x is 1 while all others are zero. By definition, the surfer's distribution at t = 1 is given by the probability vector xP; at t = 2 by (xP)P = xP^2, and so on. So $x_i = xP^i$, where $x_i(j)$ denotes the probability surfer visits page j at step i. We can thus compute the surfer's distribution over the states at any time, given only the initial distribution and the transition probability matrix P.

In our analogy, there's a situation under which the visit frequency of pages converges to a fixed frequency, called **steady-state quantity**. According to this intuition we can set Pagerank of each node v to this steady-state frequency. A vector x is a steady-state distribution iff xP = x, or iff x is the principal left eigenvector of P; so the N entries of x contains Pagerank values for the corresponding web pages. Thus if we were to compute the principal left eigenvector of the matrix P, the one with eigenvalue 1, we would have computed the Pagerank values. However, from computational point of view, it's too expensive compute Pagerank finding the eigenvector.

2 BACKGROUND

We use the *power iteration* method to compute Pagerank. This method simulates the surfer's walk: begin at a state and run the walk for a large number of steps t, keeping track of the visit frequencies for each of the states. After a large number of steps t, these frequencies "settle down" so that the variation in the computed frequencies is below some predetermined threshold ϵ. PageRank can be computed either iteratively or algebraically.

2.1 Iterative method

At the first step, k = 0, every web page has a fixed Pagerank, and that is: $PR^0(p_j) = \frac{1}{N} \forall j \in [1...N]$. While for a generic iteration k > 0, the Pagerank will be computed with the following formula:

[1] https://en.wikipedia.org/wiki/Markov_chain

$$PR^{k+1}(p_i) = \frac{1-d}{N} + d \sum_{p_j \in IN(p_i)} \frac{PR^k(p_j)}{L(p_j)} \tag{1}$$

where

- d is the *damping factor*, the probability to follow an outgoing link (instead of using teleport);
- N is the number of nodes in the web graph;
- $IN(p_i)$ is the set of all pages having a outgoing link toward p_i;
- $PR^k(p_i)$ is the Pagerank of page p_i at iteration k;
- L(p_j) is the number of outgoing links from page p_j;

The computation ends when for some small ϵ this condition is satisfied:

$$|PR^{k+1}(p_i) - PR^k(p_i)| < \epsilon \tag{2}$$

2.2 Algebraic method

We solve the following equation:

$$X = (dP + \frac{1-d}{N}E)X = AX \tag{3}$$

where

- E is a N x N matrix with all entries equal to 1;
- P is the transition probability matrix (as described in Section1.1);
- X is the probability distribution of the web surfer, represented as N-dimensional vector;
- N and d are the same as in Section1.2.1.

To solve this equation we start with an arbitrary vector x(0), then in succession we obtain $x(t+1) = Ax(t)$; the computation stops when $|x(t+1) - x(t)| < \epsilon$, for some small ϵ.

3 METHODS OF SOLUTION

In this project has been chosen the **Iterative method**(2.1) to solve the problem, because of its computational simplicity. Since the Pagerank is executed in a distributed environment, in this project will be used the Hadoop framework to compute it. However programming in *Hadoop* means define a certain number of *jobs*, each *job* represents a MapReduce phase, and the *mapper* and *reducer* functions; so it's quite different from the sequential and parallel models.
In this case must be defined an iterative job, that is a job having as input file the output file generated from itself; so the types of *key* and *value* are the same both in input and output files. This job implements one Pagerank's iteration: for this reason each row of the input and output files must contain infos to compute (1) for a page p. These infos are the Pagerank of all pages linking to p and its old Pagerank used to check termination. So the syntax of input/output file's row has the form:

$$"p \quad PR^k \quad < OutgoingList >" \qquad (*)$$

where PR^k is the Pagerank of p at k-th iteration and $< OutgoingList >$ is the list of all pages linked by p. Obviously each row is associated to a unique page.
The idea is to split each row during the map phase in order to associate to each page $s \in <$ $OutgoingList >$ the Pagerank PR^k; consequentially each page s is a *key*, in map and reduce phases, and will have as *value* the Pagerank of all ingoing pages, only in reduce phase, and its own Pagerank and outgoing list, in both the phases. An example is here:

$$"A \quad PR(A) \quad B, C" \quad --> \begin{cases} "B \quad PR(A)/2" \\ "C \quad PR(A)/2" \\ "A \quad PR(A) \quad B, C" \end{cases}$$

In the reduce phase each *key* (page) has as *value* either the <OutgoingList> and the old Pagerank or all the ingoing pages Pageranks (divided by the length of their lists) and these infos are used to compute (1) and to check the convergence condition (2). The job doesn't make another loop if all pages satisfies the convergence condition.
The input dataset *BerkStan.txt* has a specific format, different from the input/output job format (*). For this reason it is defined another job: it takes *BerkStan.txt* as input and generates an output with the format (*). So in this project are used two concatenated jobs, that is the first job's output becomes the second job's input, described following:

- preliminary, it is the first job executed and prints the input dataset in format (*), used by the second job, and makes two other output files concerning the in/out degree, used in evaluation phase;
- iterative, it's the second job and represents a generic iteration of Pagerank.

4 IMPLEMENTATION DETAILS

In this section are described how the jobs are implemented.

4.1 Preliminary Job

The main goal of this job is to scan the (*BerkStan.txt*) file and then generates a file having a biunique correspondence between a page and a row, where the syntax is described in (*). The syntax of a row in *BerkStan.txt* has this format:

$$"p_j \quad p_k"$$

which means that p_j has an hyperlink towards p_k. A row could be a comment, in that case the row starts with '#': these rows are discarded. An example of map and reduce phases in this job are represented in Fig.2, applied on *webgraph* in Fig.1.

In *preliminary* job the InputFormat, which specifies how the input file's content is split, is set up to *TextInputFormat*; so each row corresponds to the *value*. The mapper (defined in *Job1Map.java*) checks if a row is a comment (if it contains '#' or if it has more than two words or only one) or a cycle (a page links to itself); if the row is syntactically correct, for a generic row "A B" will be generated the two rows "A B" and "B $A". The second row is printed to prove the existence of page B, in fact could happen that a page has ingoing links but no outgoing link; moreover this row tells B has an in-link from A. In this job neither combiner nor partitioner are defined.
At the end the reducer (defined in *Job1Reduce.java*) takes for every page (i.e. the *key*) the corresponding outgoing list (i.e. the *value*) and filters it: if the *value*'s first char is $ (ingoing link) then the *value* (except first char) will be inserted in page *key*'s IngoingList, otherwise it will be inserted in *key*'s OutgoingList. Moreover all duplicate elements in OutgoingList and IngoingList are discarded. After filtering, *preliminaryjob* makes 3 outputs (using *MultipleOutputs*[2]):

- one concerning the *iterative* job: this is the input file used by *iterative* job at the beginning, having the format (*);
- one concerning the input degree distribution: this file has the format "$A \quad in-degree(A)$" for a generic page A;
- one concerning the output degree distribution: this file has the format "$A \quad out-degree(A)$" for a generic page A.

The first output contains as many rows as pages, with the following syntax:

$$"p \quad \frac{1}{N} \quad < OutgoingList >"$$

[2]https://hadoop.apache.org/docs/r3.0.1/api/org/apache/hadoop/mapreduce/lib/output/MultipleOutputs.html

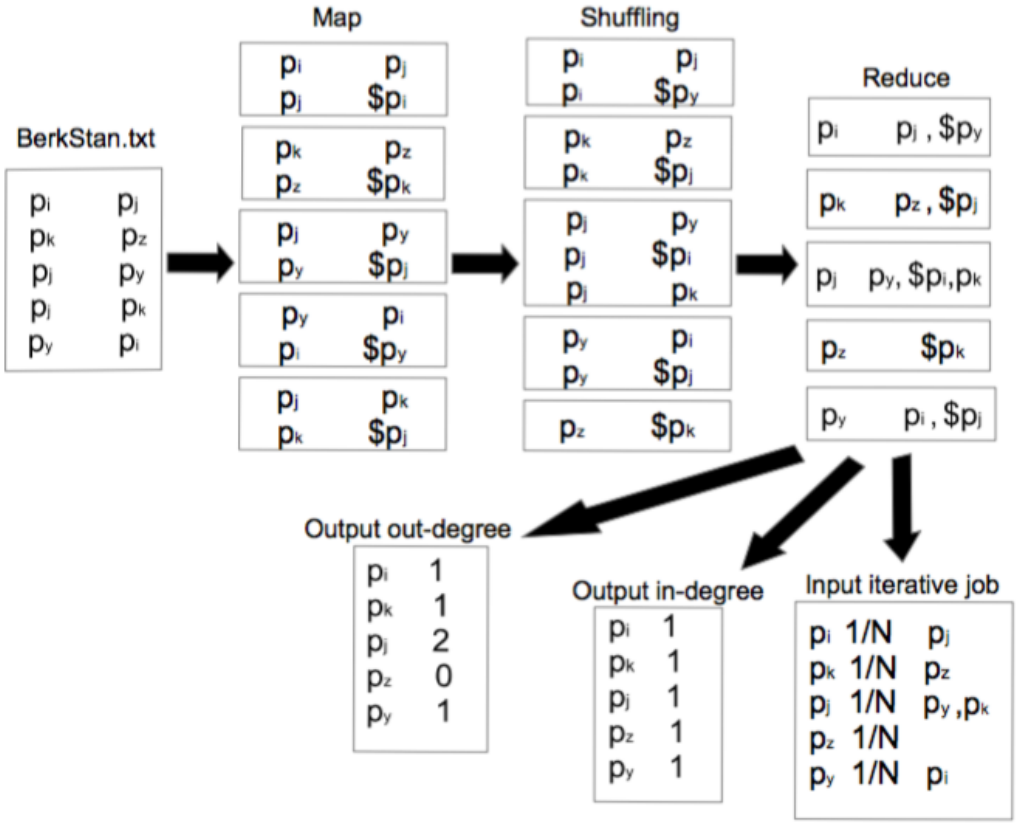


Fig. 2. MapReduce scheme in Preliminary job

this corresponds to the first step of Pagerank, because the Pagerank's initial value is $\frac{1}{N}$ for each page. Note that in *BerkStan.txt* the number of pages, N, is 685230.
The second and third output contain as many rows as pages and will be used to make evaluations concerning the in-degree and out-degree distribution. For this purpose is defined a further job, *degree* job: it takes these files and generates an output containing for each row the degree (in or out) and the number of pages having that degree.

4.2 Iterative Job

This job takes care of taking the result of the previous iteration as input and then generates the respective output, which could become the input for the next iteration. Precisely for this reason the syntax of the input and output files has to be the same, so they are equal to the syntax of the output of the preliminary phase. An example of map and reduce phases, related to a generic iteration t on *webgraph* in Fig.1, are shown in the example in Fig.3.

In this job the $KeyValueTextInputFormat$[3] is used to split the contents of the input file, i.e. the *key* becomes the page in the first column of the row while the remainder, Pagerank and outgoing list, represents the *value*. The mapper (defined in *Job2Map.java*), given a generic row of the type "$A \quad PR(A) \quad B, C$", generates two types of rows:

[3]https://data-flair.training/blogs/hadoop-inputformat/

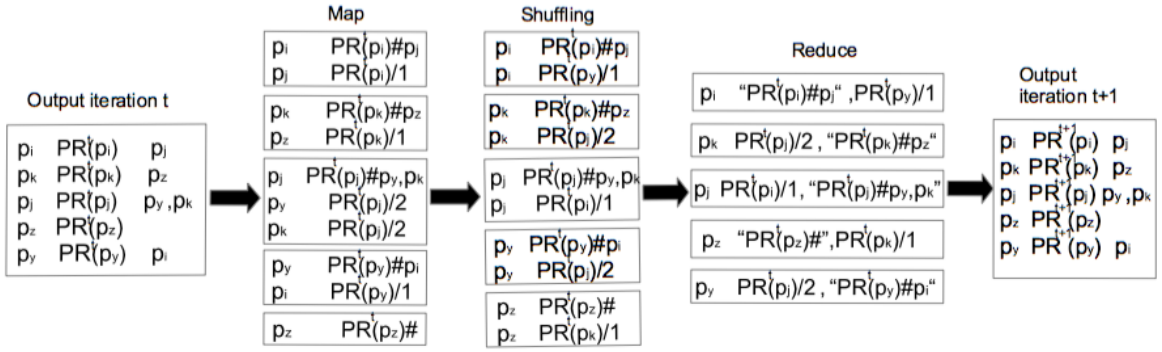


Fig. 3. MapReduce scheme in iterative job

- one row is the copy of the input row, with the '#' character added between the Pagerank and the outgoing list; in the example it is "$A \quad PR(A)\#B, C$" (if the list is empty then we will have "$A \quad PR(A)\#$");
- as many rows as there are elements in the outgoing list of the page, where each of these rows will have in the first column one of the pages in outgoing list and in the second column the Pagerank (divided by the number of elements in the list). In the example are generated "$B \quad PR(A)/2$" and "$C \quad PR(A)/2$".

Neither combiner nor partitioner have been defined. The reducer (defined in *Job2Reduce.java*) takes for each page (*key*) the associated iterable list of Text (*value*) and scans it: if an element contains '#' then the old Pagerank and the outgoing list are stored, otherwise the element is the Pagerank of a page p that points to *key* (divided by the number of elements in outgoing list of p) and this will be summed to the others Pagerank present in this kind of row.

Once the final sum is obtained, formula (1) is applied to obtain the updated Pagerank for page *key*. Then, having both the old and new Pagerank, the condition (2) is checked for every page: if all pages satisfy it then the algorithm terminates. Keep in mind that Hadoop use share-nothing paradigm. For this reason to take into account the number of pages that satisfy (2), an **Hadoop Counter**[4] [5]is used: defined using the *COUNTER.java* (enum) class and declared as *CONDITION_SAT*, and this Counter is set at each iterative job via the configuration. So if for a page p the condition (2) is satisfied then the *CONDITION_SAT* will be incremented; if at the end of an iterative job execution the *CONDITION_SAT* assumes value 685230 then the Pagerank algorithm stops, not carrying out more cycles on this job for which the output of the job will be used as the final result.

4.3 Degree Job

Its goal is to take one of the two files containing the in and out degree of each page and then generates as output the in or out degree distribution of the dataset. In this job the InputFormat is set up to *TextInputFormat*. In the map phase (defined in *DegMap.java*) each input row has the format "$p \quad degree(p)$" for a generic page p and $degree(p)$ refers either for p's in-degree or out-degree. The map phase generates as output a row "$degree(p) \quad 1$", both LongWritable. Neither combiner nor

[4]https://hadoop.apache.org/docs/r2.7.0/api/org/apache/hadoop/mapred/Counters.html

[5]An example of usage of a counter is here https://diveintodata.org/2011/03/15/an-example-of-hadoop-mapreduce-counter/

partitioner have been defined.
In reduce phase (defined in *DegReduce.java*) the $value$ contains as many 1's as there are the pages having the (in/out-)degree equal to key. In this phase all the 1s are summed and the final result will be the number of pages with that degree. So an output's row will have this format:

$$"deg\ n_{deg}"$$

5 EVALUATIONS

The project's goal is to make analysis on a real dataset: it was chosen the web graph contained in *Berkstan.txt*[6] file, collected by Berkley and Stanford universities. Analysis on *Berkstan.txt* were carried out before the project was executed on the dataset. It has been noticed that all pages have ID $\in [1, 685230]$ and that all the IDs are present in this dataset, i.e. all the pages have at least one ingoing or outgoing link.
Once the code is implemented, the goal becomes to test it. For this reason the project was carried out on appropriately written input dataset, of which the size is smaller than *BerkStan* but the values of Pagerank are known. In this file are tested different situations: self cycles, duplicate rows, pages with out-degree or in-degree equal to zero. All these cases are managed successfully. After this phase we will proceed to the evaluation phase.
The evaluations are carried out by the **Apache Commons**[7] Java library; to use it you need to put it on hdfs and to link it via the *CLASSPATH*[8] variable in execution of project. This library allows us to use some *descriptive statistics* on a set of doubles, such as mean, standard deviation, percentile, etc. For our purpose are selected three kinds of evaluations:

a) (in/out-)degree distribution;
b) statistics on Pageranks collected at each iteration;
c) statistics on convergence at each iteration.

a) The in-degree and out-degree are obtained from the *Degree* job (section 4.3). The dataset's degree distribution is represented in Fig.4: as expected both the distributions believe as a power law, for this reason the plot has been represented on a log-log scale. The maximum value for out-degree is 249 (reached by pages 514788 and 292640), while for in-degree is 84208 (reached by page 438238). We can see that there are many pages with high in-degree compared to those with high out-degree; this means that ingoing links are more distributed than outgoing links. So intuitively the Pagerank varies a lot between pages, because Pagerank depends by ingoing list.

b) The statistics chosen to describe Pageranks behavior are those offered by the Apache Commons library: precisely the mean, the max, the min and the standard deviation are chosen for this purpose. These statistics were collected in the driver: at the end of each iteration the driver reads, on HDFS, the files produced by the iterative job associated with this iteration and extracts all the computed pageranks. These will then be inserted into an instance of the *DescriptiveStructure*[9]. Then when all the pageranks have been inserted, the desired values (min, max, mean and standard deviation) are extracted. Eventually these data are inserted as String into an ArrayList, having as many elements as there are iterations. At the end of the last iteration all data in the ArrayList are written to the **csv** file.
Data were taken from the execution of the project with $\epsilon = 10^{-8}$, having 72 iterations; the other

[6] http://snap.stanford.edu/data/web-BerkStan.html
[7] https://commons.apache.org/
[8] A clear example of usage is described here https://dzone.com/articles/using-libjars-option-hadoop
[9] http://commons.apache.org/proper/commons-math/javadocs/api-3.3/org/apache/commons/math3/stat/descriptive/DescriptiveStatistics.html

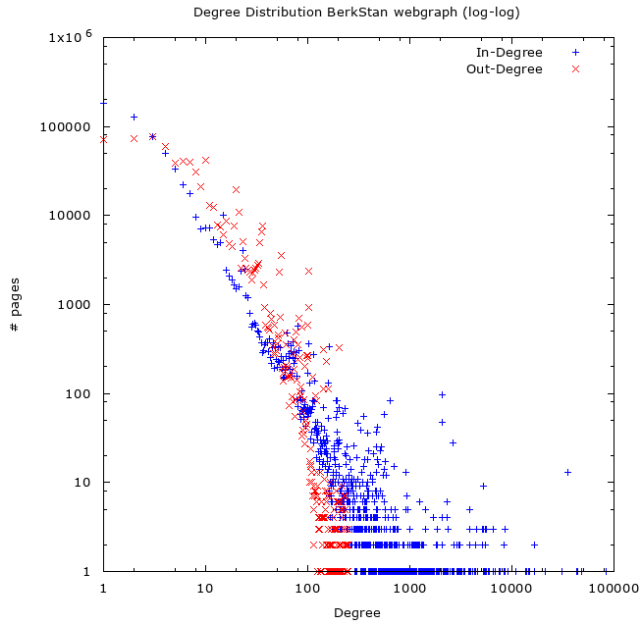


Fig. 4. In-degree (blue) and Out-degree (red) Distributon

executions, with lower values of ϵ, have the same values but less iterations.
Each column of this file corresponds to one of the four statistics listed before. In Fig.5 are described the maximum and minimum of Pagerank, while in Fig.6 are described the mean and standard deviation. From the four figures it is clear that in the initial iterations there is a lot of instability, however from the twentieth iteration this instability disappears and all the observations become stable. It can be noted that the maximum continues to vary unlike the minimum, as the number of iterations increases. This means that the minimum is the Pagerank of a page with low in-degree (which converges with a high probability immediately), while the maximum is the Pagerank of a page with high in-degree (whose convergence results with a high probability very slow).
Also the mean and variance have the same behavior as the maximum. This means that most of the pages, which have few in-links, have stabilized the Pagerank while a few pages, those with many in-links, continue to influence these measures because of their high value.

c) Finally are shown evaluations regarding the convergence speed of the Pagerank algorithm when the parameter ϵ changes. To be precise the algorithm has been executed on 5 different values of ϵ and the number of iterations made before termination was taken into account. At each iteration has been kept the number of pages satisfying the convergence condition. The Fig.7 shows the different trends.

As we can see the horizontal asymptote is 685230 (the total number of pages in dataset); as said previously, the algorithm stops when all the pages in the dataset satisfies the convergence condition. The Pagerank of a page depends mainly by its ingoing list, see (1), and from a) it can be deduced that there are a lot of pages with low in-degree and very few with high in-degree.
So in the initial iterations, a page with low in-degree satisfies immediately the convergence condition, with high probability; instead a page with higher in-degree will converge more slowly,

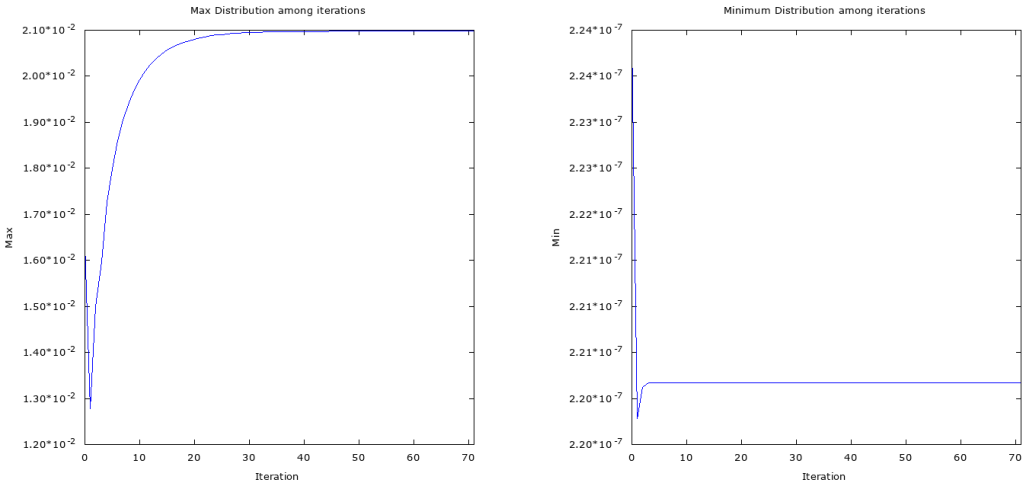


Fig. 5. Maximum and Minimum of all Pageranks collected at each iteration

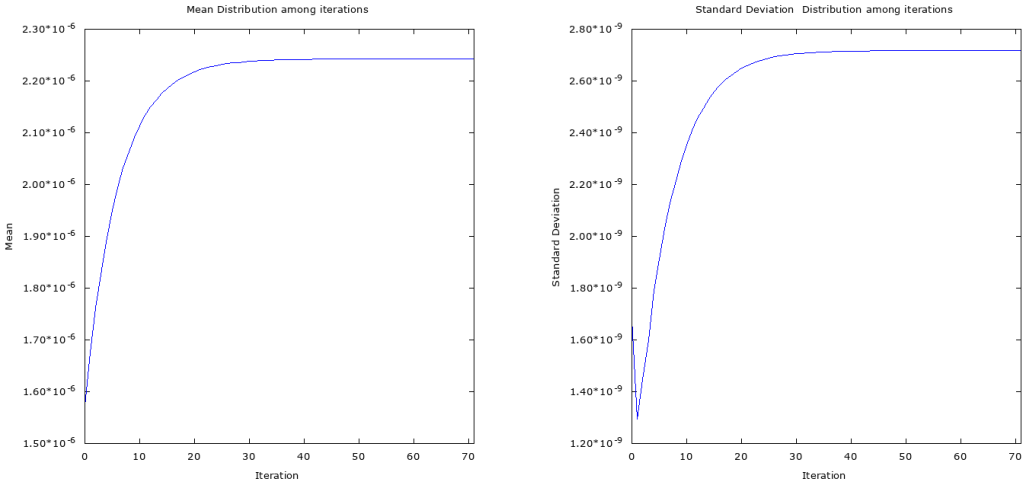


Fig. 6. Mean and Standard Deviation of all Pageranks collected at each iteration

because the Pagerank of pages in its ingoing list change and from (1) the condition is not satisfied with high probability. It can be noticed carefully, when varying ϵ, that in the last iterations the number of convergent pages slows down a lot when it is closer to 685230: this means that there are, probably, strongly connected components containing pages with high in-degree (pages with high in-degree point to similar pages) and the Pagerank of these pages changes a lot frequently.

6 CONCLUSION

In this paper concepts concerning Pagerank have been introduced and how it can be implemented in a distributed environment on a real dataset, the **Berkeley-Stanford** dataset, through the **Hadoop**

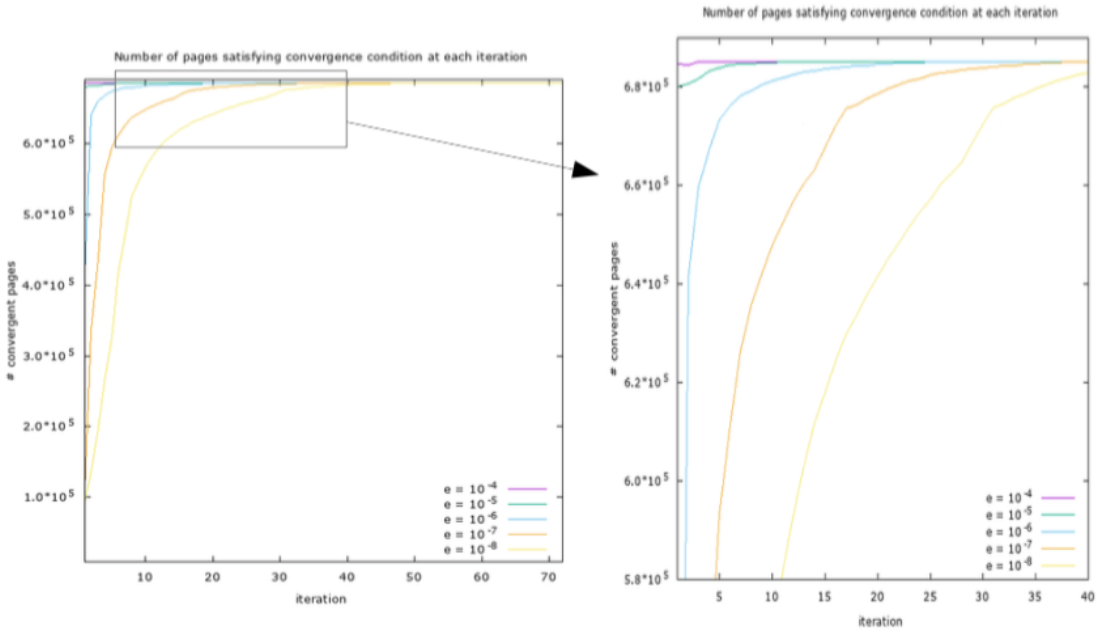


Fig. 7. Speed of convergence when the parameter ϵ changes

framework. Obviously one of many Pagerank implementations was chosen, based on the **iterative method**, which, despite its simplicity, guarantees convergence. In **Hadoop** the only way to make iterations is using *iterative* job (section 3); in this project it is sufficient to implement one iterative job to make iterations. At each iteration statistics are collected, concerning Pageranks computed in this iteration and results are described in section 5.
From observations in execution, the Pagerank convergence depends on ϵ in (2) and the convergence's speed also varies with the variation of ϵ: the value of ϵ is inversely proportional to the speed of convergence. As expected, it has been shown that the distribution of the nodes degrees follows the power law. Moreover from the results we note that the nodes with high in-degree are connected to each other, so tend to form a clique.