UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Cloud Computing Project

PAGE RANK

Cancello Tortora Giuseppe
Casini Mirko
Lagna Andrea
Macrì Armando

Academic Year: 2020-21

Summary

1. Introduction

The PageRank algorithm or Google algorithm was introduced by Larry Page, one of Google's founders. It is not the only algorithm used by Google to order search engine results, but it is the first algorithm that was used by the company, and it is the best-known. It was firstly used to rank web pages in the Google search engine.

PageRank is used to give each page a relative score of importance and authority by evaluating the quality and quantity of its links.

PageRank works by counting the number and quality of links to a page to determine a rough estimation of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

Each link from one page to another one casts a so-called vote, the weight of which depends on the weight of the pages that link to it. We can't know their weight till we calculate it, so the process goes in circle.

Each time we run the calculation, we get a closer rough estimation of the final value. We remember each calculated value and repeat the calculation several times till the obtained values stop changing so much.

In order to prevent some pages from having too much influence, the PageRank formula also uses a damping factor. According to the theory, there's an imaginary surfer who is randomly clicking on links, gets bored at some point, and stops clicking. The probability that this person will continue clicking at any step is a damping factor. In the formula, the total value of pages is damped down by multiplying it by 0.85 (a generally assumed value).

It's also considered that the average sum of all pages equals one. Thus, even if a page has no backlinks (i.e., no votes), it still gets a small score of 0.15 (one minus a damping factor).

The Web is viewed as a directed graph (the Web Graph) $G = (V, E)$, where each of the N pages is a node and each hyperlink is an arc.

The intuition behind this model is that a page $i \in V$ is "important" if it is pointed by other pages which are in turn "important".

Formally, the PageRank P of a page n is defined as follows:

$$P(n) = \alpha \frac{1}{N} + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

where:

- N is the total number of nodes in the graph
- a is the random jump factor
- $L(n)$ is the set of pages that link to n
- $C(m)$ is the out-degree of node m (the number of links on page m).

1.1. Assumption

The dataset must store Wikipedia pages according to the xml format. Each page of Wikipedia is represented in XML as follows:

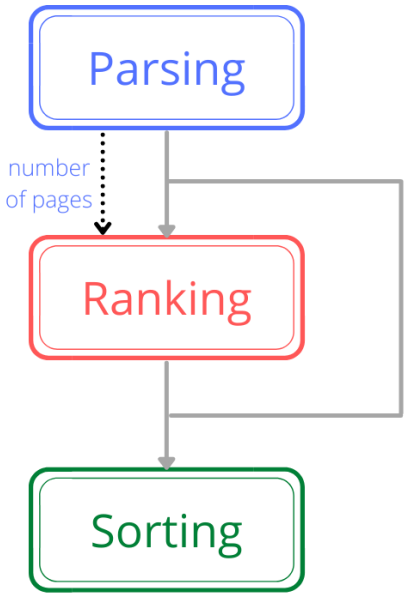
```
<title>web page name</title>
. . .
<revisionoptionalVal="xxx">
        . . .
        <textoptionalVal="yyy">page content</text>
        . . .
</revision>
```

We suppose that in the file each page is on a single line and that therefore the same is not repeated in other lines.

2. Hadoop

In this section it is described the PageRank algorithm implemented with Hadoop framework.

The work has been divided into three different Hadoop jobs: *parsing*, *calculating*, and *sorting*.



At each stage all the results are saved in different directories and the final result can be retrieved in the PageRank directory.

2.1. Parsing job

In the first job, the information in the input file is parsed and only the meaningful information is extracted, i.e., the title of the page and the outgoing links. In the map phase, after extracting the information, we emit the title and, if present, the outgoing links of that specific page.

In this job, the reducer is useless since we decided to emit directly from the map phase the value 1.0 and the link to other pages for each wikipage.

```
1  Procedure ParserMap(key, line)
2      hCounter <- hCounter + 1
3      title <- getTitle(line)
4      outgoingLinks <- getOutgoingLinks(line)
5      for each link in ourgoingLinks do:
6          EMIT (title, link)
```

2.2. Calculating job

The calculation step uses the output of the previous step to assign to each page its own page rank. The map phase emits for each page both all the outlinks with their initial ranks and the page structure composed by, of course the title, the value 1.0 and the outlinks. This information is important because it's important to maintain the page structure.

The reducer for each title collects and sum up all the contributions provided by the other pages.

Repeat these actions for more accurate results.

```
procedure setup(context)
    numPages <- context.getLong()

procedure map(key, [title + rank + outgoingLinks])

    if rank == 1.0 then
        rank <- 1/numPages
    end

    if outgoingLinks is empty then
        EMIT (title, rank)
    else
        EMIT (title, [initial rank + outgoingLinks]
    end

    for each link in outgoingLink do:
        outgoingLinkRank <- rank/outgoingLink.length
        EMIT (link, outgoingLinkRank)

procedure setup(context)
    numPages <- context.getLong()
    alpha <- context.getFloat()

procedure reducer(title, outgoingLinks[rank + link])
    pageRank <- 0.0
    for each i in range(outgoingLinks)
        pageRank <- pageRank + outgoingLinks[i].rank

    if outgoingLinks is not empty then
        val <- (1 - alpha) * pageRank + (alpha / numpages)
        EMIT(title, [val + outgoingLinks])
    end
```

2.3. Sorting job

The last job consists of sorting all the nodes with respect to their PageRank in a descending order. To do so, we implemented a WritableComparable class, called Comparator, in order to reverse the ascending order of the values.

```
procedure sortMapper(key, [title + rank + outgoinglinks])
    Comparetor.setTitle(title)
Comparator.setRank(rank)
    EMIT(Comparator, null)

procedure sortReducer(Comparator, null)
    EMIT(Comparator.getTitle(), Compartor.getRank())
```

2.4. Performance evaluation 1.0

For the Hadoop implementation, we used a customized **WritableComparable** class, called *Comparator,* used by the SortMapper and the SortReducer, through which we were able to sort the results in a descending order.

The **setup** method was used to read from the job context some information to get parameters useful for the computation.

A **combiner** is exploited during the ranking phase to reduce the quantity of intermediate data produced. The *PageRankCombiner* reduce the reducer input data from 47224 to 34209 elements with a reduction about 27,5%. It allows to reduce the amount of data moved across the network.

In the other stages combiners are useless. In the parse stage we don't use any reducer because we aggregate the information with respect to the key directly in the mapper, while in the sort stage the shaffle and sort mechanism is the main operation needed, aggregation is not used here.

A very important parameter, essential to be able to evaluate the page rank is the total number of pages considered. It makes no sense to introduce an additional step in which pages are counted, this operation can be done while parsing the pages. According to what has been said in the 'assumptions' section, it is sufficient to count how many lines are present in the input file, hence it's enough to count how many are the inputs of the map phase of the first job. This information is easily obtained through a counter managed directly by the framework.

Another important result can be obtained by acting on the number of **reducers** involved in the computation. Each job requires to specify the number of reducers but for the application it is enough to modify the number of reducers in the ranking job, since in the other jobs it is needed only one reducer.

The cases analysed to evaluate the performance, increasing level of parallelism, take into consideration four variables:

- Map Task Failed
- Total time spent by all map tasks (ms)
- Total time spent by all reduce tasks (ms)
- Execution time

Each test is evaluated with 3 iteration and alpha set to 0,15.

	Map Task Failed	Map time (ms)	Reduce Time (ms)
Iteration 0	0	3.009	2.766
Iteration 1	0	3.109	2.702
Iteration 2	0	3.056	2.910

TABLE 2.1 TEST WITH 1 REDUCER

	Map Task Failed	Map time (ms)	Reduce Time (ms)
Iteration 0	0	3.234	5.238
Iteration 1	0	5.767	5.352
Iteration 2	0	9.977	9.977

TABLE 2.2 TEST WITH 2 REDUCERS

	Map Task Failed	Map time (ms)	Reduce Time (ms)
Iteration 0	0	3.099	15.337
Iteration 1	2	40.666	27.612
Iteration 2	2	49.328	57.828

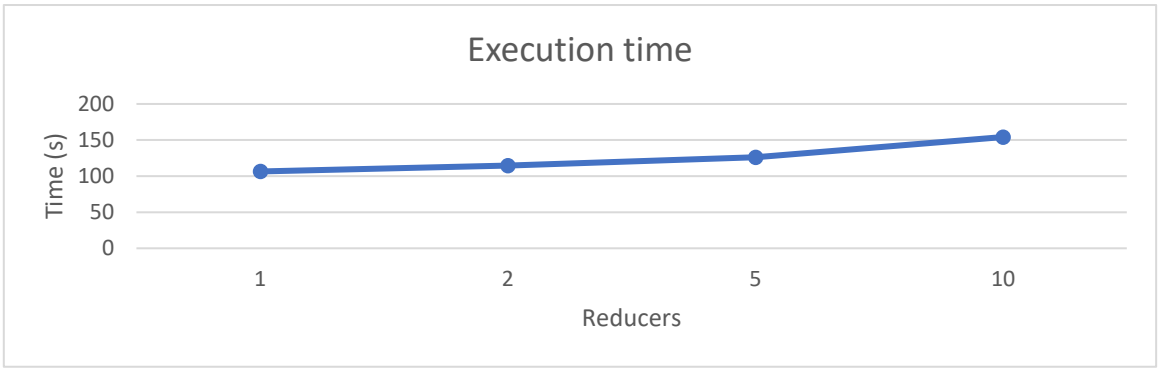
TABLE 2.3 TEST WITH 5 REDUCERS

	Map Task Failed	Map time (ms)	Reduce Time (ms)
Iteration 0	0	3.083	41.215
Iteration 1	2	113.067	158.707
Iteration 2	2	170.555	209.224

TABLE 2.4 TEST WITH 10 REDUCERS

The results reported in the tables above show how the execution of the ranking phase takes longer as the reducers increase and it leads to an increase of the total execution time.

The tests were carried out on a small dataset so at each iteration are created multiple very tiny files whenever the number of reducers increase. This mechanism introduces more I/O operations and more communication across the network, decreasing the application performances.



2.5. Performance evaluation 2.0

The second performance evaluation we performed consists in finding the execution time by changing the number of iterations. We started by using 1 reducer and we computed the execution time with 5, 10 and 15 iterations. Then, we did the same computation by using 3 reducers. As shown in the figures below, the time increases with the increase of the iterations and, of course, of the reducers.

	5 iterations	10 iterations	15 iterations
1 REDUCER	153.336 s	267.361 s	379.055 s
3 REDUCERS	200.478 s	340.638 s	521.273 s

3. Spark

In this section it is illustrated and described all the code used to implement the PageRank algorithm in Spark in both code languages: Python and Java.

3.1. Introduction and main stages

The goal of this project is to compute the page rank value for some pages provided into an input file. All the pages provided are formatted in the same way and they represent Wikipedia pages. From the input file we can retrieve the structure of the graph (due to the fact that all the outgoing links of a page are listed in the file) that is needed to compute all the page rank values. In order to calculate all the scores several steps, illustrated below, are needed. The steps are the following:

1. Read the input file and obtain all the titles and the related outlinks
2. Compute the initial page rank values as 1/N (where N is the number of nodes)
3. Calculate the contribution for each outlink starting from the rank of the parent node
4. Remove the contributions of nodes that are not in the original set of nodes
5. Sum the contribution received from each node
6. Apply the page rank formula to update all the ranks
7. Go back to step 3 and repeat for the number of iterations required
8. Sort the results by descending order of page rank value
9. Output the results in a text file

4. Pseudocode

For the implementation some aspects need to be specified:

- We considered only all the pages into the tag <title> </title>. When we compute the contributions, we also generate some pages that are not part of the initial graph, so we need to remove them from the RDD of contributions.
- We considered only the links between the tag <text> </text> to avoid spurious links.
- In order to avoid loss of pages without incoming links, we assign 0 as the contribution of a node to itself.

4.1. computeRank procedure

```
1   procedure computeRank(sum s)
2       return (1/n)*alfa + (1-alfa)*s
```

4.2. distributeContribution procedure

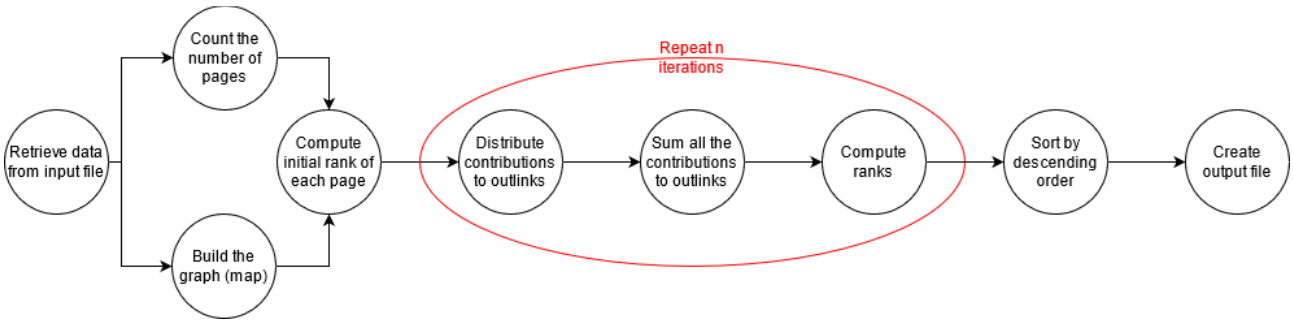
```
1   procedure distributeContribution(title t, outlinks o, rank r)
2       outlinkRank <- [(t,0)]
3       numberOutlink <- o.count()
4       if numberOutlink is greater then 0 then
5           contribution <- r/numberOutlink
6           for all outlinks in o do
7               outlinkRank.append(o, contribution)
8       return outlinkRank
```

4.3. PageRank procedure

```
1   procedure PageRank(inputFile i, outputFile o, iterations iter, dampingFactor d)
2       inputFile <- i
3       outputFile <- o
4       iterations <- iter
5       alfa <- d
6       RDDInput <- getFromInput(i)
7       n <- RDDInput.count()
8       titles <- decodeInput(RDDInput)
9       ranks <- (title, 1/n)
10      for i in range(iter) do
11          distributeRank <- join(titles, ranks).distributeContribution(title, outlinks, rank)
12          consideredRank <- distributeContribution.filter(title is in titles)
13          sumRanks <- consideredRank.sumByKey()
14          ranks <- sumRanks.computeRank(rank)
15      sort <- sumRanks.sortDescendingBy(value)
```

5. DAG

In the image below we represent the DAG containing all the main operations needed to perform the PageRank algorithm. Some of the DAG node can be break down into different actions.



6. Optimizations and performances

6.1. Caching

For what concerns the improvement of the performances, the cache() method was exploited to store the RDD graph in memory. As we can see from the pseudocode, we need to retrieve the graph structure at each iteration. This is a static structure (does not change after the first computation) so it can be saved in memory and retrieved every time it is needed.

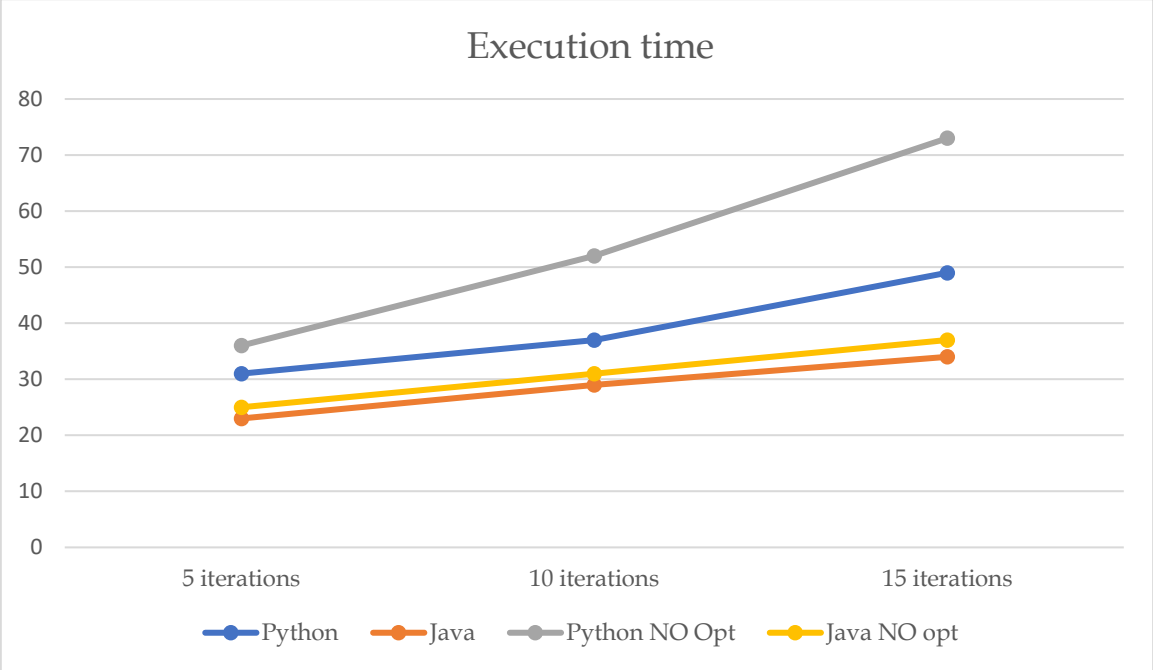
6.2. Broadcast variables

There are some variables that can be considered as read only, so we decided to cache them in memory instead of sending a copy at each computation. The first one is the variable "considered_keys_broadcast" used to remove the pages that are not part of the list of nodes contained in the tag <title> </title>. The other one is the variable "number_nodes_broadcast" used in the computation of the PageRank value.

6.3. Performance analysis

In the figure below, it is possible to see the execution time with the different implementations. Both the implementations, Java and Spark, have been tested either with the optimizations or without.
In Python, as shown in figure, the code containing the implementations takes almost half the time with respect to the version without optimizations.
But generally, there are not significant changes if we use the Python or Java versions of the code.

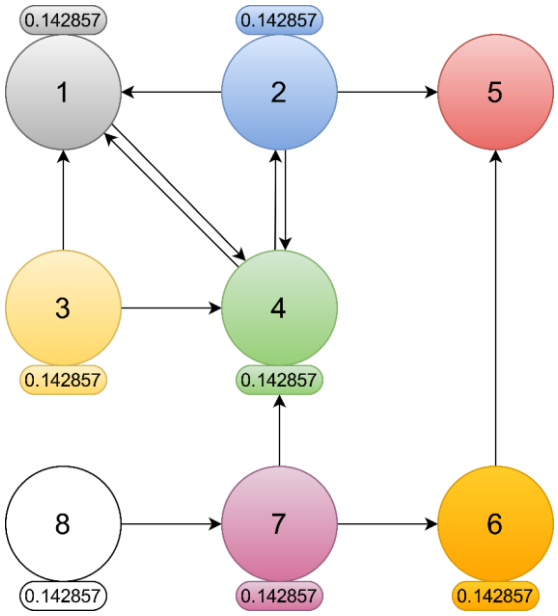
	5 iterations	10 iterations	15 iterations
Spark – Python	31.555 s	37.481 s	49.139 s
Spark – Java	23.806 s	29.056 s	34.162 s
Spark – Python no optimization	36.643 s	52.359 s	73.796 s
Spark – Java no optimization	25.721 s	31.202 s	37.457 s



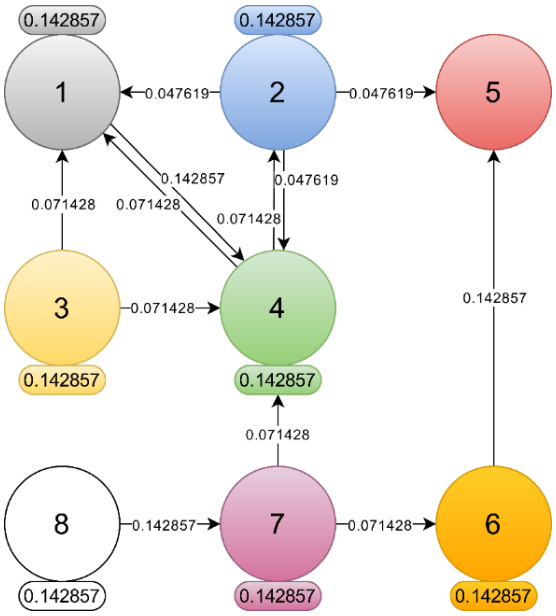
7. Test

For the analysis of the correctness of the application, we used a custom dataset containing a simplified version of the graph with different composition of link in order to simulate possible combinations.

As we can see, node 8 is what in literature is commonly referred to as disconnected component, while node 5 is an example of dangling node.

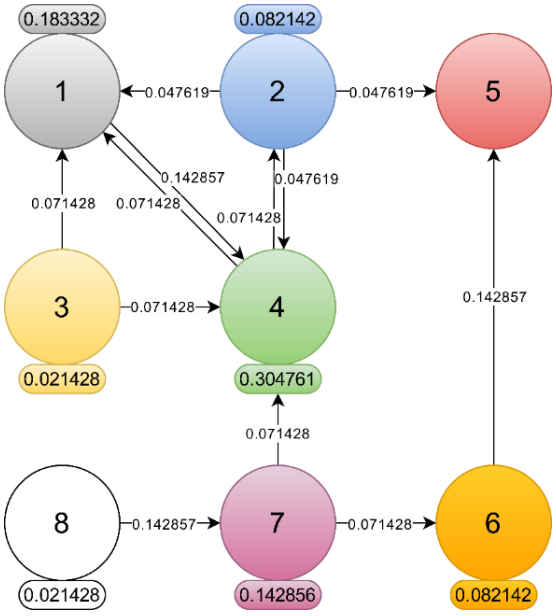


7.1. First iteration



CONTRIBUTION

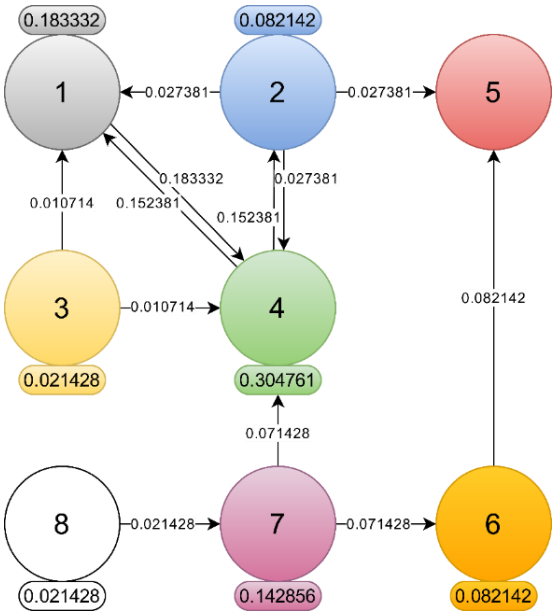
```
('4', 0)
('1', 0.07142857142857142)
('2', 0.07142857142857142)
('3', 0)
('1', 0.07142857142857142)
('4', 0.07142857142857142)
('6', 0)
('5', 0.14285714285714285)
('7', 0)
('4', 0.07142857142857142)
('6', 0.07142857142857142)
('1', 0)
('4', 0.14285714285714285)
('8', 0)
('7', 0.14285714285714285)
('2', 0)
('1', 0.047619047619047616)
('4', 0.047619047619047616)
('5', 0.047619047619047616)
```



RANK

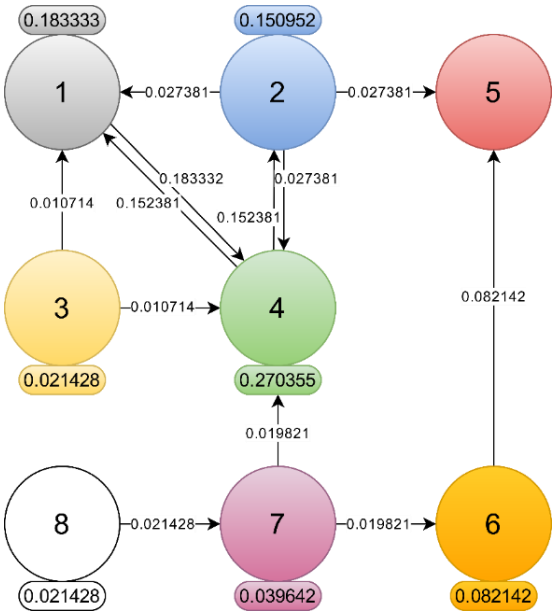
```
('4', 0.30476190476190473)
('1', 0.18333333333333332)
('7', 0.14285714285714285)
('6', 0.08214285714285713)
('2', 0.08214285714285713)
('3', 0.021428571428571425)
('8', 0.021428571428571425)
```

7.2. Second iteration



CONTRIBUTION

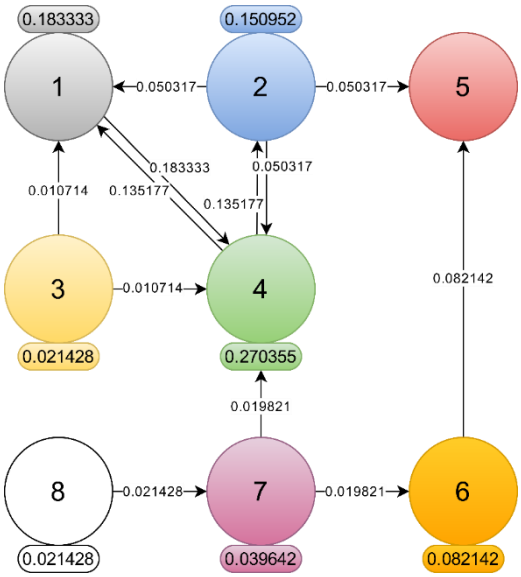
```
('4', 0)
('1', 0.15238095238095237)
('2', 0.15238095238095237)
('3', 0)
('1', 0.010714285714285713)
('4', 0.010714285714285713)
('2', 0)
('1', 0.027380952380952377)
('4', 0.027380952380952377)
('5', 0.027380952380952377)
('1', 0)
('4', 0.18333333333333332)
('8', 0)
('7', 0.021428571428571425)
('6', 0)
('5', 0.08214285714285713)
('7', 0)
('4', 0.07142857142857142)
('6', 0.07142857142857142)
```



RANK

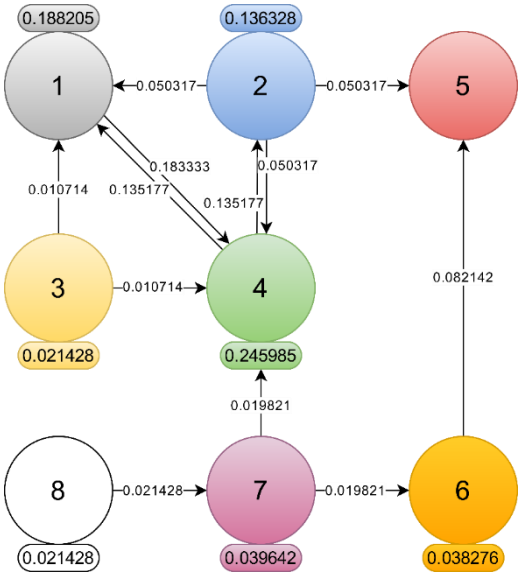
```
('4', 0.2703571428571428)
('1', 0.1833333333333333)
('2', 0.15095238095238095)
('6', 0.08214285714285713)
('7', 0.03964285714285713)
('3', 0.021428571428571425)
('8', 0.021428571428571425)
```

7.3. Third iteration



CONTRIBUTION

```
('4', 0)
('1', 0.1351785714285714)
('2', 0.1351785714285714)
('7', 0)
('4', 0.019821428571428566)
('6', 0.019821428571428566)
('1', 0)
('4', 0.1833333333333333)
('8', 0)
('7', 0.021428571428571425)
('3', 0)
('1', 0.010714285714285713)
('4', 0.010714285714285713)
('6', 0)
('5', 0.08214285714285713)
('2', 0)
('1', 0.050317460317460316)
('4', 0.050317460317460316)
('5', 0.050317460317460316)
```



RANK

```
('4', 0.24598710317460312)
('1', 0.18820734126984123)
('2', 0.1363303571428571)
('7', 0.03964285714285713)
('6', 0.03827678571428571)
('8', 0.021428571428571425)
('3', 0.021428571428571425)
```

In the picture above it is illustrated an example of the PageRank algorithm with 3 iterations. The pictures on the left represent graphically the iterations with the computations done by hand, instead on the right we have all the outputs, both rank and contributions, produced by the application.