PageRank

Cloud Computing final project - Prof. Nicola Tonellotto

Leonardo Turchetti
Lorenzo Tonelli
Ludovica Cocchella
Rambod Rahmani

Msc. in Artificial Intelligence and Data Engineering

June 5, 2021

Contents

1 Introduction

The importance of a web page is an inherently subjective matter, which depends on the readers interests, knowledge and attitudes. PageRank can be defined as a method for rating web pages objectively and mechanically, effectively measuring the human interest and attention devoted to them. In order to measure the relative importance of web pages, PageRank was proposed as a method for computing a ranking for every web page based on the graph of the web.

The project focused on designing a MapReduce algorithm (using pseudocode) to implement the PageRank (using both Hadoop and Spark). Initially, a pseudocode implementation and the design assumptions are presented. The successive sections focus on the implementation details using both Hadoop and Spark. Finally, the validation results

obtained using both a realistic and a synthetic dataset are provided.

The entire codebase is available at https://github.com/lorytony/CloudComputing.

You can access the VM with the source code ready to be executed using `ssh hadoop@172.16.3.218` providing the password `sicurezza34!`. The project files can be found under ∼`/PageRank/hadoop` and ∼`/PageRank/spark`.

2 PageRank

PageRank[1] is a measure of web page quality based on the structure of the hyperlink graph. Although it is only one of thousands of features that is taken into account in Google's search algorithm, it is perhaps one of the best known and most studied. Every page has some number of forward links (out-edges) and backlinks (in-edges). We can never know whether we have found all the backlinks of a particular page but if we have downloaded it, we know all of its forward links at that time.

The reason why PageRank is so interesting is that there are many cases where simple citation counting does not correspond to our common sense notion of importance. For example, if a web page has a link off the Yahoo home page, it may be just one link but it is a very important one. This page should be ranked higher than many pages with more links but from obscure places. PageRank is an attempt to see how good an approximation to "importance" can be obtained just from the link structure.

The previous example models the so called "Propagation of Ranking Through Links": a page has high rank if the sum of the ranks of its backlinks is high. This covers both the case when a page has many backlinks and when a page has a few highly ranked backlinks.

Formally, given

- a page p_i among the total N nodes (pages) in the graph;
- the set of pages $L(p_i)$ that link to p_i;
- and the out-degree $C(p_j)$ of node p_j;
- the random jump factor α;

the PageRank PR of a page p_i is defined as follows:

$$PR(p_i) = \alpha \frac{1}{N} + (1 - \alpha) \sum_{p_j \in L(p_i)} \frac{PR(p_j)}{C(p_j)}$$

The definition of PageRank above has another intuitive basis in random walks on graphs. The simplified version corresponds to the standing probability distribution of a random walk on the graph of the Web. Intuitively, this can be thought of as modeling the behaviour of a "random surfer". The "random surfer" simply keeps clicking on successive links at random. However, if a real Web surfer ever gets into a small loop of web pages, it is unlikely that the surfer will continue in the loop forever. Instead, the surfer will jump

[1] The PageRank Citation Ranking: Bringing Order to the Web - January 29, 1998 - http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

to some other page. The additional factor α can be viewed as a way of modeling this behaviour: the surfer periodically "gets bored" and jumps to a random page. This residual probability, α, is usually set to 0.15, estimated from the frequency that an average surfer uses his or her browser's bookmark feature. Alternatively, $1 - \alpha$ is referred to as the "damping" factor.

2.1 Computation

PageRank can be computed either iteratively or algebraically. Using the iterative method, at $t = 0$, an initial probability distribution is assumed

$$PR(p_i, 0) = \frac{1}{N}$$

where N is the total number of pages, and $(p_i, 0)$ is page i at time 0. At each time step, the computation, as detailed above, yields

$$PR(p_i, t+1) = \alpha \frac{1}{N} + (1 - \alpha) \sum_{p_j \in L(p_i)} \frac{PR(p_j, t)}{C(p_j)}.$$

2.2 Implementation Assumptions

In the presented implementation, some assumptions were made.

Firstly, nowadays a colossal 4.2 billion pages exist on the Web, spread across 8.2 million web servers. No crawling operations were taken into account. In what follows, the assumption was made that the the inputs to the program are pages from the Simple English Wikipedia. We will be using a pre-processed version of the Simple Wikipedia corpus in which the pages are stored in an XML format. Each page of Wikipedia is represented in XML as follows:

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

The pages have been "flattened" to be represented on a single line. The body text of the page also has all newlines converted to spaces to ensure it stays on one line in this representation. This makes it easy to use the default `InputFormat`, which performs one `map()` call per line of each file it reads. Links to other Wikipedia articles are of the form `[[page name]]`. Starting from this input file, first the hyperlink graph is constructed and then the PageRank is computed for each node.

The second assumption was related to how *dangling nodes* should be handled. Dangling nodes are pages that do not have any out-links: our random surfer will get stuck on these pages, and the importance received by these pages cannot be propagated. In the original PageRank publication by Larry Page and Sergey Brin, it is said that "*Because dangling links do not affect the ranking of any other page directly, we simply remove them from the system until all the PageRank values are calculated. After all PageRank values are calculated, they can be added back in, without affecting things significantly. Notice*

the normalization of the other links on the same page as a link which was removed will change slightly, but this should not have a large effect.". The decision was made to remove dangling nodes: they will not be visible in the links output as well.

The "jump factor" α and the number of iterations to be performed are expected as inputs. As a matter of fact, the proposed implementation does not rely upon the iterations convergence measured as the difference between consecutive PageRank values as stopping criteria. But instead a fixed number of iterations are performed.

2.3 Pseudocode Implementation

The solution we came up with is made up of multiple stages: each of them can also be thought of as a MapReduce job.

Stage 0: counts the number of nodes of the hyperlink graph

```
   Data: XML input data
   Result: Number of nodes of the hyperlink graph N
 1 class Mapper
 2     intermediate ← 0
 3     method Map(lineid k, line l)
 4         if "<title>*</title>" ∈ l then
 5             intermediate ← intermediate + 1
 6         end
 7
 8     method Cleanup()
 9         EMIT(term N, count intermediate)
10
11 class Reducer
12     method Reduce(term t, counts [c1, c2, ...])
13         sum ← 0
14         for all count c ∈ counts [c1, c2, ...] do
15             sum ← sum + c
16         Emit(term t, count sum)
```

Stage 1: builds the Hyperlink Graph

```
   Data: XML input data
   Result: Hyperlink graph with initial PageRank
 1 class Mapper
 2     method Map(lineid k, line l)
 3         if "<title>page_name</title>" ∈ l then
 4             title ← "page_name"
 5             while "<text>[[out_link]]</text>" ∈ l do
 6                 adj_list ← adj_list + "out_link"
 7             Emit(term title, list adj_list)
 8         end
```

```
1  class Reducer
2      initial_pr ← 0.0
3      method Setup()
4          N ← Hadoop.config.get("N")
5          initial_pr ← 1/N
6
7      method Reduce(term t, adj_list [o_1, o_2, ...])
8          output ← initial_pr + " "
9          for all out_link o ∈ adj_list [o_1, o_2, ...] do
10             output ← output + o
11         Emit(term t, term output)
```

Stage 2: iteratively computes the PageRank

```
   Data: Hyperlink Graph
   Result: PageRank values list
1  class Mapper
2      method Map(lineid k, line l)
3          title ← line.parse_title()
4          initial_pr ← line.parse_initial_pr()
5          adj_list ← line.parse_adj_list()
6          contribution ← initial_pr/adj_list.size()
7          for all out_link o ∈ adj_list [o_1, o_2, ...]
8              Emit(term out_link, count contribution)
9          Emit(term title, adj_list [o_1, o_2, ...])
10
11     class Reducer
12         method Setup()
13             N ← Hadoop.config.get("N")
14             alpha ← Hadoop.config.get("ALFA")
15
16         method Reduce(term t, contributions [c_1, c_2, ...])
17             sum ← 0
18             adj_list ← line.parse_adj_list()
19             for all contribution c ∈ contributions [c_1, c_2, ...] do
20                 sum ← sum + c
21             pr ← alpha · 1/N + (1 − alpha) · sum
22             output ← pr + " " + adj_list
23             Emit(term t, count pr)
```

Stage 3: sorts PageRank values list

```
   Data: PageRank values list
   Result: Sorted PageRank values list
 1 class Mapper
 2     method Map(lineid k, line l)
 3         title ← line.parse_title()
 4         final_pr ← line.parse_final_pr()
 5         Emit(count final_pr, term title)
 6
 7     class Reducer
 8         method Reduce(count final_pr, terms [t1, t2, ...])
 9         for all term t ∈ terms [t1, t2, ...] do
10             Emit(term t, count final_pr)
```

3 PageRank Implementation using Hadoop

The PageRank implementation using Hadoop was divided into 4 MapReduce Jobs:

- `job0`: in charge of counting the number of nodes in the hyperlink graph; it parses each line of the input `.xml` file extracting the content of the `<title>` tag and counting how many of such tags are found; the computed N parameter is set in the global jobs configuration;
- `job1`: in charge of building the initial hyperlink graph structure; it parses each line of the input `.xml` file extracting the content of the `<title>` and `<text>` tags; the content of the `<text>` tag is further processed in order to detect out-links formatted as `[[page name]]`; for each node, it also assigns the initial PageRank computed as $\frac{1}{N}$;
- `job2`: for each node, it calculates the PageRank contribution for each out-edge; the graph structure is preserved in order to allow for iterative scheduling of the job; the computed contributions are used to calculate a new PageRank value for each page; this job is scheduled a number of times equal to the value of the given command line argument `iterations`;
- `job3`: after the iterations of `job2` are over, this job is in charge of sorting in descending order the final PageRank results;

The Java implementation consists of the following classes:

- `Driver.java`: this class implements the Hadoop driver;
- `NodesCounterMapper.java`: the `map()` method is called once for each of the lines in the input `.xml` file; whenever a `<title>` tag is found, this is a node; the fixed key `N` is outputted with the values aggregated by an intermediate In-Mapper combiner;
- `NodesCounterReducer.java`: this reducer is used by `job0`; the `reduce()` method, simply sums the values outputted by the mapper to obtain the final count of the nodes in the hyperlink graph;

- `GraphBuilderMapper.java`: mapper for `job1`; the `map()` function is called once for each of the lines inside the input `.xml` file to extract the content of the `<title>` and the `<text>` tags; it emits as key the page name and as value the list of out-edges separated by '`]]`'; the '`]]`' separator was choosen becuase it is the only combinations of chars we are guarateed not to find within the `[[page name]]`;
- `GraphBuilderReducer.java`: the `reduce()` method computes the initial PageRank value as $\frac{1}{N}$, and emits the page title as key and the initial PageRank value followed by the graph structure as output value; `n1` 0.2 `n3]]n4`;
- `PageRankMapper.java`: this Mapper is used by `job2`; the `map()` method is called once for each of the lines of the output generated by `job1`; for each line, the out-links are extracted and for each of them the incoming contributions are computed; each outlink and the received contribution is emitted as the Key-Value pair; additionally also the graph structure is emitted;
- `PageRankReducer.java`: the `reduce()` method is called once for each outlink and sums the received contributions in order to be able to compute the new PageRank;
- `SorterMapper.java`: parses the output produced by `job2` to remove out-links and emit the page title as key and the PageRank as value; title and PageRank are switched in this case in order to be able to apply the MapReduce sorting process;
- `SorterReducer.java`: it emits the list of pairs `<key,value>` in descending order;
- `DescendingDoubleWritableComparator.java`: this class implements the WritableComparator used to sort in descending order the output generated by SorterReducer.

4 PageRank Implementation using Spark

The PageRank implementation using Spark was written using Python.

```python
import re
import sys
from operator import add
from pyspark import SparkContext

__author__ = "Leonardo Turchetti, Lorenzo Tonelli, Ludovica Cocchella and Rambod Rahmani"
__copyright__ = "Copyright (C) 2021 Leonardo Turchetti, Lorenzo Tunelli, Ludovica Cocchella ... and Rambod Rahmani"
__license__ = "GPLv3"

def parseLine(line):
    """
        Called once for each line of the input .xml file.
        Parses the <title></title> and <text>[[]]</text> tags
        to retrieve the page title and the outlinks.
    """
    title = re.findall(r'<title>(.*?)</title>', line)
    text = re.findall(r'<text.*>(.*?)</text>', line)
    outlinks =  re.findall(r'\[\[([^]]*)\]\]', text[0])
    return title[0], outlinks

def countContributions(outlinks, pageRank):
    """
        Computes the size of the given outlinks list and
        returns the contribution to each outlink.
    """
    count = len(outlinks)
    for link in outlinks:
        yield(link, pageRank/count)

if __name__ == "__main__":
    # parse command line arguments
    iterations = int(sys.argv[1])
    alfa = float(sys.argv[2])
    inputFile = sys.argv[3]

    # connect to the Hadoop cluster
    master = "yarn"
```

```
38      sc = SparkContext(master, "PageRank")
39
40      # create input file RDD
41      inputRDD = sc.textFile(inputFile)
42
43      # build hyperlink graph: graph = list(K, V), K=title[0], V=[outlinks]
44      graph = inputRDD.map(lambda line: parseLine(line))
45
46      # remove dangling nodes
47      graph = graph.filter(lambda node: len(node[1]) > 1).cache()
48
49      # count the number of rows in the input RDD (nodes)
50      N = graph.count()
51
52      # broadcast the number of nodes to the workers
53      broadcastN = sc.broadcast(N)
54
55      # compute initial pageranks
56      # pageRanks = list(K, V), K=title, V=initialPageRank
57      pageRanks = graph.map(lambda node: (node[0], 1/float(broadcastN.value)))
58
59      # compute pageRank iteratively
60      for iteration in range(iterations):
61          # completeGraph = list(K, V), K=title, V=[[outlinks], initialPageRank]
62          completeGraph = graph.join(pageRanks)
63
64          # contributions = list(K, V), K=outlink, V=contribution
65          contributions = completeGraph.flatMap(lambda token: countContributions(token[1][0], ...
                token[1][1]))
66
67          # compute new PageRank value, pageRanks = list(K, V), K=outlink, V=PageRank
68          pageRanks = contributions.reduceByKey(add, 10).mapValues(lambda sum: ...
                alfa*(1/float(broadcastN.value)) + (1 - alfa)*sum)
69
70          # handle disconnected nodes in the hyperlink graph
71          disconnectedNodes = graph.map(lambda node: (node[0], ...
                alfa*(1/float(broadcastN.value)))).subtractByKey(pageRanks)
72          pageRanks = pageRanks.union(disconnectedNodes)
73
74      # retrieve graph structure: no dangling nodes admitted
75      filteredPageRanks = graph.join(pageRanks)
76      filteredPageRanks = filteredPageRanks.map(lambda node: (node[0], node[1][1]))
77
78      # order nodes with descending pagerank order
79      pageRanksOrdered = filteredPageRanks.sortBy(lambda a: -a[1])
80
81      # save ordered pagerank results as text file
82      pageRanksOrdered.coalesce(10, True).saveAsTextFile("spark-output")
83
84      # stop spark context
85      sc.stop()
```

The lines we think need additional comments are 70 - 72: this is how we handle disconnect components in the graph. Because these links are not directly linked by any other node in the hyperlink graph, they are removed on line 68 when contributions are computed (they do not receive contributions). After all the PageRanks are calculated, they can be added back in without affecting things: they preserve their initial PageRank computation.

Also, on line 75 we are basically removing all those nodes generated when computing contributions. At the very beginning, the assumption was made that the hyperlink graph nodes were only those pages found within the `<title></title>`: this assumption was kept throughout both the Java (Hadoop) and Python (Spark) implementations.

Finally, on lines 68 and 82 the number of tasks to be used and the number of final partitions is forced to 10: this was done because the number of tasks/partitions obtained on the synthetic dataset reached value 59, while on the real world dataset even more than 8000 tasks/partitions were created. Too many tasks/partitions will generate excessive overhead in managing many small tasks. The value 10 was considered to be a good trade-off between computation parallelization and overhead.

On the `namenode` machine, the Spark implementation can be executed simply by running:

```
hadoop@namenode:$ cd PageRank/spark/
hadoop@namenode:/PageRank/spark$ spark-submit pagerank.py 3 0.15 UltimateTest.txt
```

5 Validation

Validation was performed using both a synthetic and a real word dataset.

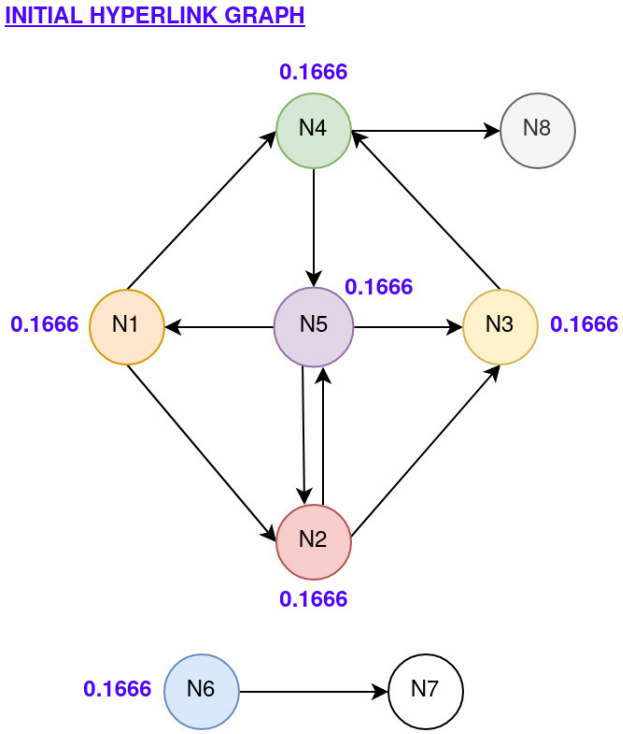
5.1 Synthetic dataset

Validation was performed on a test dataset containing a sample hyperlink graph with dangling nodes and disconnected graph components as well.

```
<title>n1</title><revision><text> content [[n2]],[[n4]] </text></revision>
<title>n2</title><revision><text>[[n3]],[[n5]] </text></revision>
<title>n3</title><revision><text attr="val">[[n4]] content </text></revision>
<title>n4</title><revision><text>[[n5]] content [[n8]] </text></revision>
<title>n5</title><revision><text attr="val">[[n1]],[[n2]],[[n3]] </text></revision>
<title>n6</title><revision><text attr="val"> content [[n7]] </text></revision>
<title>n8</title><revision><text attr="val"> content</text></revision>
```

As we can see, node `n6` is what in literature is commonly referred to as *disconnected component*, while node `n7` and node `n8` are examples of *dangling nodes*.

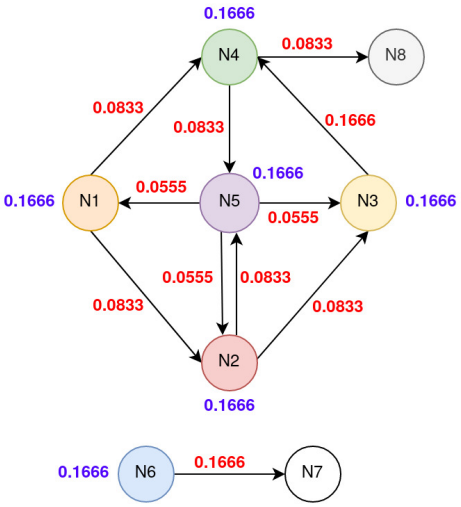
The following pictures display 3 iterations of the PageRank algorithm performed manually (pen, paper and calculator) on the synthetic dataset with $\alpha = 0.15$. Underneath each iteration, also our implementation output is reported:

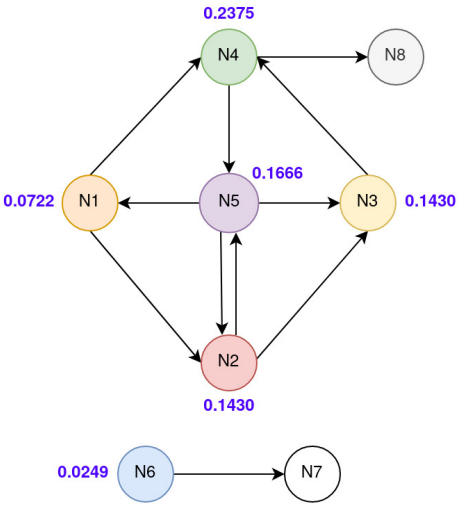


Initial PageRank

```
('n1', 0.16666666666666666)
('n2', 0.16666666666666666)
('n3', 0.16666666666666666)
('n4', 0.16666666666666666)
('n5', 0.16666666666666666)
('n6', 0.16666666666666666)
```

ITERATION 1





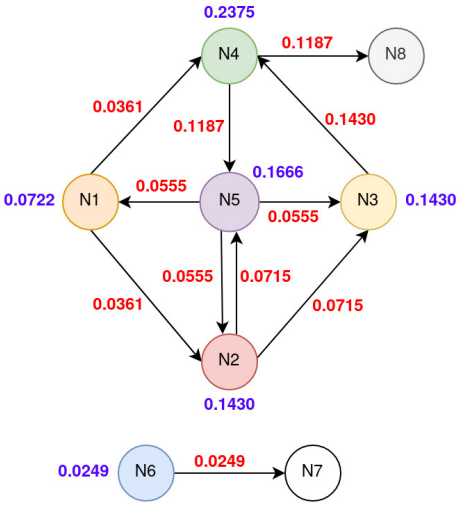
Contributions

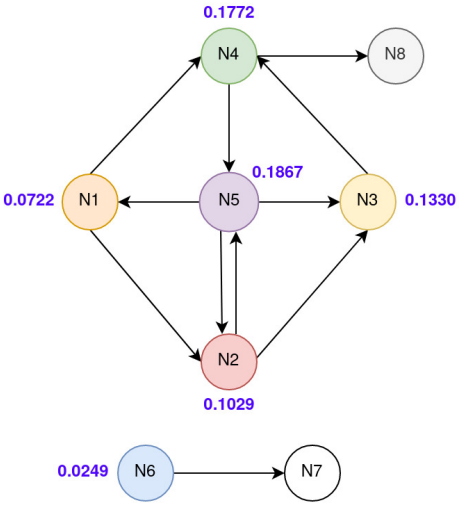
```
('n2', 0.08333333333333333)
('n4', 0.08333333333333333)
('n1', 0.05555555555555555)
('n2', 0.05555555555555555)
('n3', 0.05555555555555555)
('n7', 0.16666666666666666)
('n5', 0.08333333333333333)
('n8', 0.08333333333333333)
('n3', 0.08333333333333333)
('n5', 0.08333333333333333)
('n4', 0.16666666666666666)
```

New PageRanks

```
('n5', 0.16666666666666666)
('n8', 0.09583333333333333)
('n1', 0.07222222222222222)
('n4', 0.2375)
('n2', 0.14305555555555555)
('n3', 0.14305555555555555)
('n7', 0.16666666666666666)
('n6', 0.024999999999999998)
```

ITERATION 2



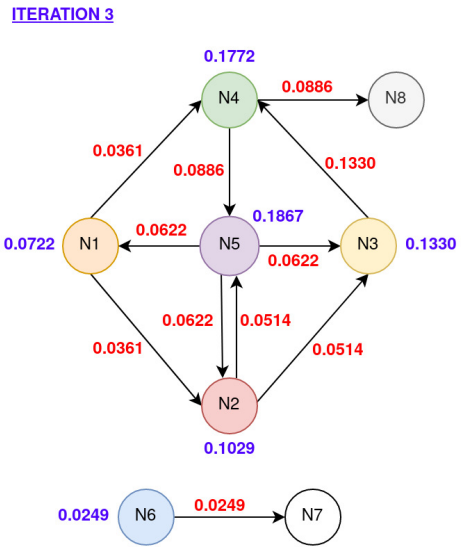


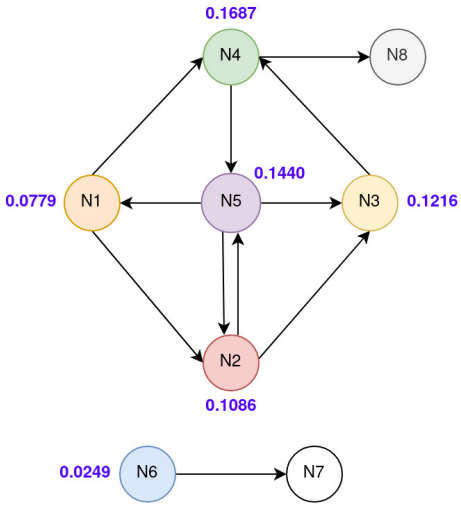
Contributions

```
('n7', 0.024999999999999998)
('n3', 0.07152777777777777)
('n5', 0.07152777777777777)
('n1', 0.05555555555555555)
('n2', 0.05555555555555555)
('n3', 0.05555555555555555)
('n5', 0.11875)
('n8', 0.11875)
('n4', 0.14305555555555555)
('n2', 0.03611111111111111)
('n4', 0.03611111111111111)
```

New PageRanks

```
('n8', 0.1259375)
('n7', 0.04625)
('n2', 0.10291666666666666)
('n5', 0.1867361111111111)
('n4', 0.17729166666666663)
('n3', 0.13302083333333334)
('n1', 0.07222222222222222)
('n6', 0.024999999999999998)
```





Contributions

```
('n5', 0.08864583333333331)
('n8', 0.08864583333333331)
('n2', 0.03611111111111111)
('n4', 0.03611111111111111)
('n4', 0.13302083333333334)
('n1', 0.06224537037037037)
('n2', 0.06224537037037037)
('n3', 0.06224537037037037)
('n3', 0.05145833333333333)
('n5', 0.05145833333333333)
('n7', 0.024999999999999998)
```

New PageRanks

```
('n4', 0.16876215277777779)
('n7', 0.04625)
('n8', 0.10034895833333331)
('n1', 0.07790856481481481)
('n3', 0.12164814814814813)
('n5', 0.14408854166666665)
('n2', 0.10860300925925924)
('n6', 0.024999999999999998)
```

Once both implementations and the debugging stage was concluded, the first and foremost validation we performed consisted in comparing the PageRank computation results produced by the Hadoop and Spark implementations. Having obtained the same exact output on this synthetic test dataset is by no means a rigorous validation, however it is a beginning.

5.2 `wiki-micro.txt` dataset

Finally both implementations were executed on the given `wiki-micro.txt` real world dataset. This allowed for observing the convergence of the PageRank computation. In an

execution with 10 iterations on a hyperlink graph made up of 2282 nodes, convergence was reached after a few iterations. This is consistent with what found in the original PageRank paper: "*on a* 322 *million link database convergence to a reasonable tolerance in obtained in roughly* 52 *iterations*".