UNIVERSITÀ DI PISA

University of Pisa

Laurea Magistrale (MSc) in Artificial Intelligence and Data Engineering

Project

Cloud Computing

Page-Rank

Gaetano Niccolò Terranova 506349

Giacomo Piacentini 627555

Rossella De Dominicis 623902

GITHUB: https://github.com/gaetanoterra/PageRank

Academic year 2020-2021

Sommario

Introduction

PageRank is a measure of web page quality based on the structure of the hyperlink graph. A PageRank results from a mathematical algorithm based on the webgraph, created by all World Wide Web pages as nodes and hyperlinks as edges. The rank value indicates an importance of a particular page. A hyperlink to a page count as a vote of support. The PageRank of a page is defined recursively and depends on the number and PageRank metric of all pages that link to it. A page that is linked to by many pages with high PageRank receives a high rank itself.

It is only one of thousands of features that is taken into account in Google's search algorithm, but it is perhaps one of the best known and most studied.

PageRank relies on the uniquely democratic nature of the web by using its vast link structure as an indicator of an individual page's value. In essence, Google interprets a link from page A to page B as a vote, by page A, for page B. But, Google looks at considerably more than the sheer volume of votes, or links a page receives; for example, it also analyzes the page that casts the vote. Votes cast by pages that are themselves "important" weigh more heavily and help to make other pages "important." Using these and other factors, Google provides its views on pages' relative importance.

PageRank Algorithm

Formally, given

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

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

The random jump factor α is sometimes called the "teleportation" factor; alternatively, (1- α) is referred to as the "damping" factor. PageRank can be computed either iteratively or algebraically. Using the iterative method, at initial time

$$P(n) = \frac{1}{N}$$

Implementation

The inputs to the program are pages from the Simple English Wikipedia. Each page of Wikipedia is represented in XML as follows:

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

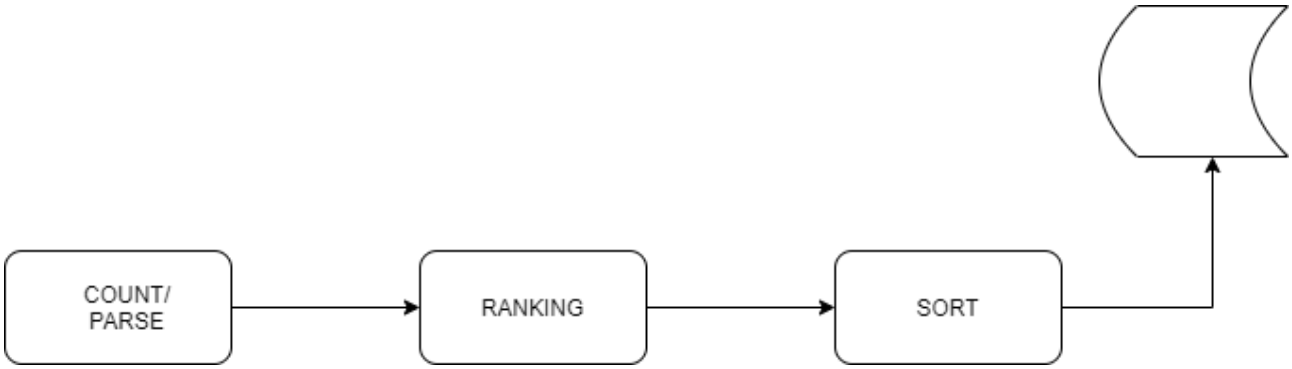
Dangling nodes are pages that do not have any out-links, so we decide to remove dangling nodes until all the PageRank values are calculated, because they do not affect the ranking of any other page directly.

The α is fixed to 0,15 and the number of iterations is asked to the client; in this documentation we performed 10 iterations.

Pseudocode Implementation

There are 3 different stages of the Algorithm:

1. Count/Initial Rank
2. Ranking
3. Sort



1. **Count/Initial Rank**

```
class CountAndRankLinksEmitter
    method PAGENUMBER()
        page_number <- 0
        file <- open(input_file)
        while rows in file
            pagenumber++
        return pagenumber
```

```
class COUNTMAPPER
    method MAP(id i, line l)
        for lines in line[l1, l2, ...]
            title <- getTitle
            outlinks <- getOutlinks
            if outlinks is not null
                for link in outlinks[o1, o2, ...]
                    EMIT(title, link)
            else
                EMIT(title, null)

    class COUNTREDUCER
        method REDUCE(title t, link l)
            page_number <- getConfig("page_number")
            page_rank <- 1/page_number
            output <- page_rank
            for links in link[l1, l2, ...]
                output <- output + ":::" + links
            EMIT(t, output)
```

```
titolo:::1/N:::link:::link
```

2. ***Ranking***

```
class RankingCalculation
    class RankingCalculationMapper
        method MAP(id i, line l)
            token = l.split(":::")
            title <- token[0]
            page_rank <- "page_rank:" + token[1]/(len(token) -2)
            for t in token[t2, t3, ...]
                links <- t[i]
                EMIT t, page_rank
            EMIT title, links
```

```
class RankingCalculationReducer
    method REDUCE(pageid i, values v)
        page_number <- getConfig("page_number")
        alpha <- getConfig("alpha")
        page_rank <- 0
        links <- ""
        for val in values[v1, v2, ...]
            if (val contains "page_rank:")
                page_rank <- page_rank + val
            else
                links <- val
        page_rank <- page_rank + alpha*(1/page_number) + (1 - alpha)*page_rank
        output <- page_rank + ":::" + links

        EMIT i, output
```

3. ***Sort***

```
class SortPage
    class SortPageMapper
        method MAP(id i, line l)
            for lines in line[l1, l2, ...]
                token <- lines.split(":::")
                page_name <- token[0]
                page_rank <- token[1]
                EMIT page_rank, page_name

    class SortPageReducer
        method REDUCE(ordered_pagerank p, pagename n)
            EMIT p, n
```

PageRank Implementation using Hadoop

Our Hadoop implementation is comprehensive of five classes, three of them running hadoop jobs: the "Parse" job which does the parsing of title and links of each page; the "RankingCalculation" job which calculate the page rank for each page; the "SortPage" job which sorts all pages in decreasing order. Every job run a single map() and a single reduce() method. On the following we describe on detail the classes we implemented:

PageRank.java: this class is the driver class, which create all the instances of CountAndRankLinksEmitter, RankingCalculation and SortPage, running them all. This class expect three arguments by the user, respectively the ***input file name***, the ***output file name***, the ***number of iterations*** for the ranking calculation phase. The alpha value is fixed to 0.15.

CountAndRankLinksEmitter.java: this class make two things: implements the pageNumber() method, that return the number of pages we have into the input file; run the map() and reduce() methods. The map() method read every line of the input file and take title and links of the page (using the getTitle() and getLinks() methods), passing them to the reduce() method which calculate a partial page rank and write into the output

file a string of the form like “page_name:::page_rank:::link1:::link2...”.

RankingCalculation.java: this class take as input the output file of Count reduce(), so the map() method read every row of the document, saving page_number, page_rank and links using a split() to separete them all. A partial page rank is then calculated with the one found in the Count class and the number of links of the page, so two couples are sent to the reduce() method: the first one is “link:::page_rank” and the second one is “page_name:::link1:::lin2...”.

SortPage.java: this class take as input the output file of RankingCalculation reduce(), reading every line of the file, which format is like “page_name:::page_rank:::links”. So the mapper read every line and split it selecting the page name and page rank and sending them to the reduce() in opposite order, so page_rank as a key and page_name as value. In this way the reduce() receive the data sorted by the page_rank (the descending order is obtained with the CompositeKeyComparator class). The reduce() write on the output file the data received.

CompositeKeyComparator.java: this class handle the sorting order of data between map() and reduce() into the SortPage class.

PageRank implementation using Spark

The implementation of Spark was written fully in python.

```
1  import re
2  import sys
3  from pyspark import SparkContext
4
5  #function that selects title and links of each node, returns => title, links
6  def findTitleAndLinks(input):
7      title = re.findall("<title>(.*?)</title>", input)
8      links = re.findall("\\[\\[(.*?)\\]\\]", input)
9
10     return title[0], links
11
12 #function that assigns an initial value for the algorithm and selects node and links, returns => title, links, initial_page_rank
13 def contributeCalculus(line):
14     page_rank = 0
15     if len(line[1]) > 0:
16         page_rank = line[2]/len(line[1])
17     else:
18         page_rank = 0
19     return line[0], line[1], page_rank
20
21 #function that associates to each link (if any) the contribution it receives, returns => title (link), contribution
22 def getLinkContribution(line):
23     if len(line[1]) > 0:
24         for link in line[1]:
25             yield link, line[2]
26     else:
27         yield line[0], 0
28 #function that computes page_rank, returns => node, page_rank
29 def rankingcalculus(line):
30     return line[0], (alpha_value/node_number + (1-alpha_value) * line[1])
```

```
31
32  #function that returns node, links and, depending on whether it is a node to which nobody points or not, the constant or the page_rank with contributions
33  def rewrite(line):
34      if line[1][1] == None:
35          return line[0], line[1][0], alpha_value/node_number
36      else:
37          return line[0], line[1][0], line[1][1]
38
39  if __name__ == "__main__":
40      input_file_name = sys.argv[1]
41      output_file_name = sys.argv[2]
42      alpha_value = float(sys.argv[3])
43      iterations_number = int(sys.argv[4])
44      sc = SparkContext("yarn", "PageRank")
45      text = sc.textFile(input_file_name)
46      #counting the nodes (pages) of the input document
47      node_number = text.count()
48      #call the findTitleAndLinks function on the document to select titles and links
49      title_links_rdd = text.map(lambda line : findTitleAndLinks(line)).cache()
50      initial_page_rank = 1/node_number
51      # I reorder the data by putting node_name, initial_page_rank
52      node_rank_rdd = title_links_rdd.map(lambda line : (line[0], initial_page_rank))
53      for i in range(iterations_number):
54          # i make a join between (node, links) and (node, page rank) obtaining -> node, (links, pagerank) -> after rewrite -> node, links, pagerank
55          node_rank_rdd = title_links_rdd.join(node_rank_rdd).map(lambda line : rewrite(line))
56          # Call the contributeCalculus function to calculate the contribution each node would make to its links
57          contribute_ranking_rdd = node_rank_rdd.map(lambda line : contributeCalculus(line))
58          # Call the getLinkContribution function to associate the contribution it receives with each link (node that have no inlinks are excluded brcause they are not links)
59          link_contribution_association_rdd = contribute_ranking_rdd.flatMap(lambda line : getLinkContribution(line))

60           # Group the nodes that have the same key (the same name) and add the partial page_ranks; then I calculate the ranking with rankingcalculus
61           somma_rdd = link_contribution_association_rdd.reduceByKey(lambda x, y : x + y).map(lambda line : rankingcalculus(line))
62           # We associate the value alpha/N to the nodes with no inlinks
63           no_inlink_nodes_rdd = title_links_rdd.map(lambda node: (node[0], alpha_value*(1/node_number))).subtractByKey(somma_rdd)
64           # We put all nodes (nodes with no inlinks and with inlinks) together
65           node_rank_rdd = somma_rdd.union(no_inlink_nodes_rdd)
66       # Sort in descending order according to rank and output node + page_rank
67       sort_rdd = node_rank_rdd.sortBy(lambda x: -x[1]).map(lambda line : (line[0], line[1]))
68       for element in sort_rdd.collect():
69           print("somma_rdd: " + str(element))
70       #save the output to a file
71       sort_rdd.saveAsTextFile(output_file_name)
72
```

The assumptions that we made to find which were the nodes of the hyperlink graph inside the documents where that the nodes are the title of the web pages contained between <title> and <\title>.
Same reasoning for the connections that were found between square brackets.
In line 53, we put an iteration cycle, in which the user can specify the number of iterations.

Functions description:

The function ***contributeCalculus:*** calculates the contribute of each node would give to its links (so node that are not a link are excluded).

getLinkContribution: it associates to each link the contribute that they get from the nodes.

rankingCalculus: it groups the nodes with same key and calculates partial PageRanks, and after it calculates the ranking.

Then we select all nodes excluded by the contributeCalculus associating all nodes with the value alpha/N and then removing nodes we have already considered in the contributeCalculus.

We make the union of the rankingCalculus and the selection made at the step before.

Finally we apply the sorting to sort_rdd: that will provide us the list of the PageRank in descending order.

At the end of we save the results of the Spark algorithm in a file named “spark-output”.

Tests

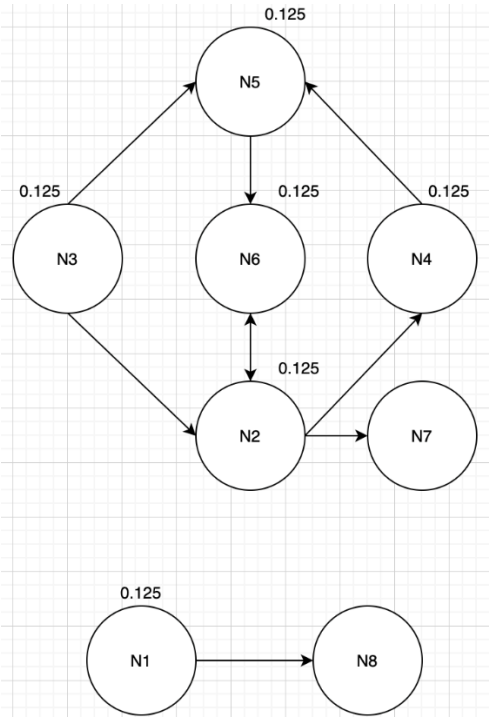
The validation was performed on a synthetic dataset that we created and a dataset that was given to us by the professor.

Synthetic dataset was made of eight different nodes, where nodes number 8 and 7 are dangling and node n1 is disconnected.

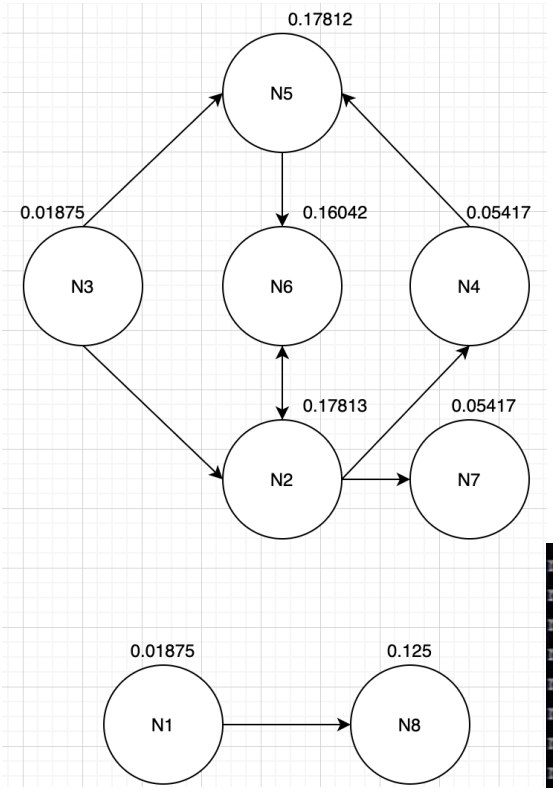
```
<title>n1</title><revision><text attr="val">[[n8]] content </text></revision>
  <title>n2</title><revision><text>[[n4]] ,[[n6]] ,[[n7]] </text></revision>
  <title>n3</title><revision><text> content [[n2]] ,[[n5]] </text></revision>
  <title>n4</title><revision><text>[[n5]] </text></revision>
  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

The pictures underneath represent the values of PageRank algorithm after 3 iterations performed with α = 0.15 which is reported to be to most common use value for that parameter.

Inizial Pagerank

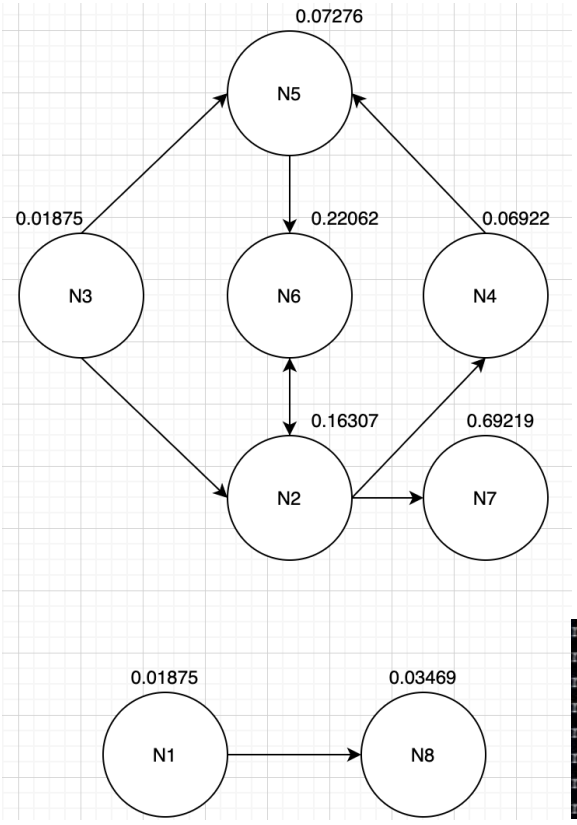


Iteration 1



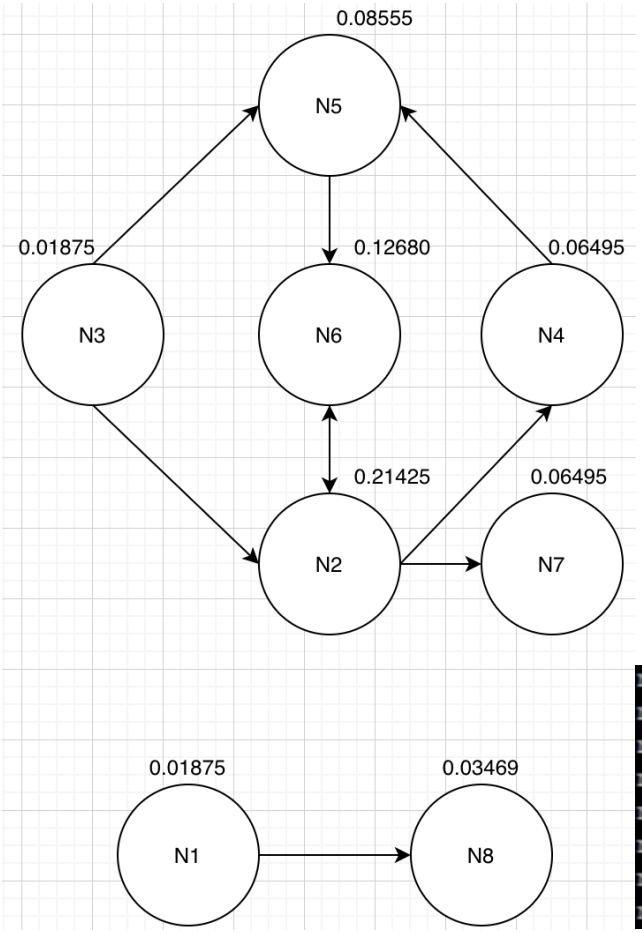

```
n1:::0.01875:::n8
n2:::0.17812499999999998:::n7:::n6:::n4
n3:::0.01875:::n5:::n2
n4:::0.05416666666666667:::n5
n5:::0.17812499999999998:::n6
n6:::0.16041666666666665:::n2
n7:::0.05416666666666667
n8:::0.125
```

Iteration 2




```
n1:::0.01875:::n8
n2:::0.16307291666666662:::n7:::n6:::n4
n3:::0.01875:::n5:::n2
n4:::0.06921875:::n5
n5:::0.07276041666666666:::n6
n6:::0.22062499999999996:::n2
n7:::0.06921875
n8:::0.034687499999999996
```

Iteration 3




```
n1:::0.01875:::n8
n2:::0.21424999999999994:::n7:::n6:::n4
n3:::0.01875:::n5:::n2
n4:::0.06495399305555553:::n5
n5:::0.08555468749999999:::n6
n6:::0.12680034722222222:::n2
n7:::0.06495399305555553
n8:::0.034687499999999996
```

As a validation method for these results, we decided to compare the results that we obtained with Hadoop and Spark, and we got the same exact results with booth methods.

The implementations of these method were finally done on the real-world dataset ‘wiki-micro.txt’ provided from the Professor that represent 2282 nodes of real-world Wikipedia pages with respective connections.

We decided to show the results after 10 iterations, in which we obtained convergence to a reasonable tolerance result after just a few iterations.

Below are the results of the Pagerank algorithm on the Simple Wikipedia database:

```
0.002215698393077872:::Special:DoubleRedirects|double-redirect
0.0015699680675731358:::Wikipedia:GFDL standardization
0.0014089263641179587:::Wikipedia:Non-free content/templates
6.747012772970742E-4:::WP:AES|←
5.338696712616357E-4:::Project:AutoWikiBrowser|AWB
4.8819675811694627E-4:::Wikipedia:Deletion review|deletion review
3.9451998351874726E-4:::user:freakofnurture|...
3.9451998351874726E-4:::Template:R from title without diacritics|R from title without diacritics
2.943301548060509E-4:::User:AWeenieMan/furme|FURME
2.7941914059687067E-4:::Wikipedia:Articles for deletion/PAGENAME (2nd nomination)
```