# Lab 8 – PageRank over Wikipedia using Giraph

CC5212-1 – May 12, 2021

Today we're going to use PageRank to find out which Wikipedia articles are "more important" and use those scores to improve the ranking of last week's search results (*ideally you have completed Lab 7 before this one, but in case you haven't, we will leave an alternative method so you can complete this lab*). To compute PageRank, we will use Apache Giraph: a distributed framework for processing graphs based on Hadoop.

As input, you are given a TSV (tab-separated values) file with two columns. This file encodes a directed graph of links between articles in Wikipedia. In more detail, for any two Wikipedia Articles A and B, there exists a link A → B in the graph if Article A contains a link to Article B. For example, the following lines in the input mean that the article on the left links to the three articles on the right:

```
http://es.wikipedia.org/wiki/Andorra   http://es.wikipedia.org/wiki/España
http://es.wikipedia.org/wiki/Andorra   http://es.wikipedia.org/wiki/Portugal
http://es.wikipedia.org/wiki/Andorra   http://es.wikipedia.org/wiki/Francia
```

We will be working with `hdfs://cm:9000/uhadoop/shared/wiki/es-wiki-links.tsv.gz`, containing 31,506,565 lines, each of which encodes a link from one Wikipedia article to another.

We will first code a PageRank algorithm to estimate the importance of articles in the graph based on how they are linked. We will then use these ranks to improve the search results of Lab 7.

- Download the code project from u-cursos and open it in Eclipse. Your initial task will be to complete `PageRank`, which computes PageRank scores over a graph loaded into Giraph. The other classes are complete: `TextNullTextEdgeInputFormat` provides code to load the graph in the aforementioned TSV format into Giraph, while `VertexValueTextOutputFormat` provides code to write the PageRank results computed by Giraph out to a TSV file. `PageRankAgg` initialises two *aggregators* that can be used to track some global information for the PageRank process. Finally `SortByRank` is a Hadoop job that will sort the output of the PageRank results for us in descending order of rank (so we can see the most important articles at the start of the file). We recommend to briefly revise these classes.
- Now to complete the PageRank algorithm. Open up the `PageRank` Java class. Most of the code is given to you. The constants are as follows:
  - `ITERS` defines the number of PageRank iterations to run.
  - `D` is the damping factor: the probability that the random surfer follows a link when on a node with outlinks.
  - `RR_AGG_NAME` is a global aggregator that keeps track of the probability of the random surfer making a random jump in the previous iteration (either from a dangling node with no outlinks, or with probability $1 - \texttt{D}$ from a node with outlinks).
  - `NUM_V_NAME` is a global aggregator to count the number of vertices. `ONE` is an object we will reuse for this count.
- The Giraph framework is based on "supersteps". In each superstep, each vertex can receive messages from the previous superstep, and can send messages for the current superstep. Within each superstep, the `compute` method is called for each vertex, along with the messages it receives. A typical superstep will use the incoming messages of a vertex to update its state, and then compute the outgoing messages sent from the vertex to its neighbours. Note that the ID of the first superstep is `0`.

- To compute PageRank, we need to know the number of vertices, but we do not have this *a priori*. While it seems trivial perhaps, given that our computation is mostly based on passing messages between neighbouring vertices, counting all vertices is not so direct. Rather we will use two initial supersteps to count the vertices in the graph. The first superstep "activates" the vertices by sending dummy messages through the graph. The second superstep then sends a 1 from each vertex to a global sum aggregator, which, as a result, will give the count of vertices at the end of the superstep.[1]
- In the third superstep, we begin the first PageRank iteration. We begin by getting the count of vertices from the aggregator, and initialise a rank for the vertex. If this is the first iteration, the rank of the vertex will be computed as the reciprocal of the number of vertices ($\frac{1}{|V|}$). Otherwise, if we are in a later iteration, the rank of the vertex will be computed as the sum of incoming messages from its neighbours, and the random jump probability divided by the number of vertices.
- The vertices of the graph can be viewed as compute nodes that do computations. If we are in the last superstep, we tell the vertices to halt and end the computation. Otherwise, if we not in the last superstep, we should send messages from the current vertex to its neighbours for the superstep that follows, and update the aggregators as necessary. In terms of what messages a vertex should send and how it should update the aggregators, there are two main cases: the current vertex has outlinks, or the current vertex does not have outlinks. There are three `TODO`s left for you:
  - In the case that the vertex has outlinks, you must (1) send messages to split a ratio of `D` of the vertex's current rank evenly with its neighbours, (2) send what remains of the vertex's current rank to the aggregator for the random jump probability. Note that in the case of (1), `sendMessageToAllEdges` will send the argument value to each of the vertex's neighbours.
  - In the case that the vertex has no outlinks, (3) send the vertex's complete current rank to the aggregator for the random jump probability.
- Once you have filled in the aforementioned three values, compile the JAR for the project, transfer it to a personal folder on the cluster.
- We have provided a small example (`hdfs dfs -cat /uhadoop/shared/pr/pr-ex.tsv`) for testing. Try to run your code on this small example (the following is all one command):[2]

  ```
  giraph mdp-giraph.jar org.mdp.hadoop.cli.PageRank
  -eif org.mdp.hadoop.io.TextNullTextEdgeInputFormat -eip /uhadoop/shared/pr/pr-ex.tsv
  -vof org.mdp.hadoop.io.VertexValueTextOutputFormat -op /uhadoop2021/GROUP-FOLDER/pr-ex
  -w 1 -ca mapreduce.job.tracker=yarn -ca mapreduce.framework.name=yarn
  -mc org.mdp.hadoop.pr.PageRankAgg
  ```

  (If you get a command not found error, run `source ~/.profile` and try again.)
- Check the output folder: `hdfs dfs -ls /uhadoop2021/GROUP-FOLDER/pr-ex/`. Have a look at the output data inside: `hdfs dfs -cat /uhadoop2021/GROUP-FOLDER/pr-ex/part-m-0000N`. The PageRank scores should sum to 1. The rank of `a` should be approximately 0.169.
- Once your code works, we want to run it on the full data. Run:

  ```
  giraph mdp-giraph.jar org.mdp.hadoop.cli.PageRank
  -eif org.mdp.hadoop.io.TextNullTextEdgeInputFormat
  -eip /uhadoop/shared/wiki/es-wiki-links.tsv.gz
  -vof org.mdp.hadoop.io.VertexValueTextOutputFormat -op /uhadoop2021/GROUP-FOLDER/pr-full
  ```

[1] Giraph does provide a `getTotalNumVertices()` method, but this returns $-1$ in the first superstep, and appears to not count dangling vertices without outlinks.

[2] The arguments indicate, respectively, the class with the `compute` method, the input format for reading the data, the input file, the output format for writing the data, the output folder, the number of workers, two configurations to run Giraph on Hadoop, and finally the class that initialises the global aggregators. It's quite long as we do not use a `main` method; rather we specify the relevant configurations in the command call.

```
-w 4 -ca mapreduce.job.tracker=yarn -ca mapreduce.framework.name=yarn
-mc org.mdp.hadoop.pr.PageRankAgg
```

We increase the number of workers (`w`) to `4` to increase parallelism. Review the output folder and the results inside. Note that the process might take some time to run.

- We want to know what the top-ranked articles are, right? For this, we will use a Hadoop job to sort the results by rank (descending). Run (all one command):[3])

  - `hadoop jar mdp-giraph.jar SortByRank -D mapreduce.job.reduces=1 /uhadoop2021/` GROUP-FOLDER `/pr-full/ /uhadoop2021/` GROUP-FOLDER `/pr-full-s/`

  While you're waiting, trying guessing what the most important article(s) in Wikipedia might be. Revise the first few results to see if you're correct!

- Did we just compute the PageRank of Wikipedia articles for fun? Naturally, yes. But now that we have them we can also improve the search results for Lab 7. Ideally we could extend the lab code for Lab 7 directly, but since the lab is still open, some may not have completed it yet. Likewise we cannot provide the code because it includes the solution for Lab 7 (we will publish the code after the deadline for Lab 7). Rather for now we provide a binary jar with the code compiled that you can use. What is most important for now is not the code, but how the search results may change when PageRank is included. (As mentioned in the class, relevance and importance are often antagonistic measures and there is no clear-cut way to combine them; we apply a simple/heuristic combination of both scores for a query that might not work super well for all searches.)

  - First, you will need to have the ranking data available locally. Call (all one command):

    * `hdfs dfs -copyToLocal /uhadoop2021/` GROUP-FOLDER `/pr-full-s/part-r-00000 /data/2021/uhadoop/` GROUP-FOLDER `/ranks.s.tsv`

  - On the server, go to `cd /data/2021/uhadoop/lab7`. You will find a jar. Call (better to use **lowercase** for NEW-INDEX-NAME as Elasticsearch can have problems with uppercase names):

    * `java -jar mdp-lab07.jar BuildWikiIndexBulkWithRank -i /data/uhadoop/shared/wiki/es-wiki-articles.tsv.gz -igz -p /data/2021/uhadoop/` GROUP-NAME `/ranks.s.tsv -o` NEW-INDEX-NAME

    This will index the same documents as Lab 7, but now with our PageRank scores as a new field.

  - Using mdp-lab07.jar, run `SearchWikiIndex` and `SearchWikiIndexWithRank` over the new index to perform searches and compare the effects without (`SearchWikiIndex`) and with boosting (`SearchWikiIndexWithRank`) based on PageRank scores:

    * `java -jar mdp-lab07.jar` SEARCH-CLASS `-i` NEW-INDEX-NAME

    In particular, `SearchWikiIndexWithRank` will retrieve the PageRank score from the relevant results and combine it with the relevance score in order to reorder the results, taking into consideration the importance of the article. `SearchWikiIndexWithRank` should, on the other hand, give the same results as Lab 7, not considering PageRank.

    Try search for "`obama`" with and without PageRank. Try five other searches of your choice. How do the results change?

- SUBMIT: (1) your `PageRank.java` class, (2) the top-10 Wikipedia articles according to PageRank, and (3) the results of five queries (one of which should be "`obama`") with and without adding PageRank.

[3]The `-D mapreduce.job.reduces=1` property sets the number of reducers to one, so we get one output file.