Lab 8 – PageRank over Wikipedia

CC5212-1

May 11, 2015

Today we're going to find out what are the most important articles in Wikipedia using PageRank (the algorithm originally used by Google to decide the most important pages on the Web ... the one that made them famous).

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
```

The main task of the lab is to code a PageRank algorithm that will compute rankings of the importance of articles in the graph based on how they are linked. Later we will use these ranks to improve the search features developed last week.

- Download the code project from `http://aidanhogan.com/teaching/cc5212-1/code/mdp-lab8.zip` and open it in Eclipse. There's a bunch of classes left in there for reference. We'll be using `OIDCompress`, `PageRankGraph`, `OIDDecompress`, and `SortByRank`. The only class we'll need to code in is `PageRankGraph`; the others are done!
- Start downloading the data from `http://aidanhogan.com/teaching/cc5212-1/data/lab8/`. You can download both files. `es-wiki-links.gz` contains 31,506,565 lines, with an article encoded in each line (*there is no need to unzip this file; it's 4GB uncompressed; we can read a GZIP input stream from the Java program later*). If you open up the second file, `es-wiki-links-10k.gz`, it contains the first 10,000 lines so you can see the format. Each line has a link like above.
- When the data are downloaded, open up the code project. Since the data are large, first we're going to do a little trick that will help us manage the data in memory: we are going to map the full URL strings in the links file to unique integer IDs. The code is already done for you in `OIDCompress`.[1] Call `OIDCompress` with

  ```
  -i [in-dir]/es-wiki-links-10k.txt -o [out-dir]/es-wiki-links-10k.oid.txt
    -d [out-dir]/es-wiki-links-10k.dict.txt
  ```

  The `-o` parameter gives the location of the output file that will contain the compressed graph. The `-d` parameter records the mapping from strings to OIDs that we will use to decompress the file later. Open both output files and have a look! The first line of the compressed file gives the number of nodes in the graph; the rest give OID-encoded links. Next call the same method for the big file:[2]

[1] It doesn't do anything fancy except load the strings in memory and assign them an ID. To avoid loading the strings in memory, we could also sort the strings using the hard-drive and assign sequential IDs, then sort the graph, join on ... it's doable but more complicated. ☺

[2] If you have trouble with memory, add, e.g., `-Xmx1500M` to the VM arguments to increase the heap.

```
-i [in-dir]/es-wiki-links.txt.gz -igz -o [out-dir]/es-wiki-links.oid.txt.gz -ogz
  -d [out-dir]/es-wiki-links.dict.txt.gz -dgz
```

- Okay our graph is now ready to be ranked. Now to the main task: code the PageRank algorithm. Open up the `PageRankGraph` Java class. There's some code already done for you to load the compressed graph into memory as an `int[][] graph`. So to get the outlinks of node i, we can access `int[] out_i = graph[i]`, for example (it will be important to remember that `out_i` may be null if i has no outlinks).
- Your task is to code the `rankGraph(int[][] graph)` method.
  - First get the number of nodes in the graph.
  - Create two arrays (`double[]`) to hold the rank of each node. The first array holds the current rank (`newranks`). The second array holds the rank from the previous round (`oldranks`).
  - We need to initialise some ranks. Into `oldranks` we need to set an initial value for every node. What might it be? *Remember that all ranks in both vectors should always add up (modulo minor rounding issues) to 1.*
  - There is a number of iterations defined at the top of the class file as `ITERS`. We need to start the iteration loop (`for(int i=0; i<ITERS, i++){ ... }`).
  - From the last lecture, hopefully you remember that there are two types of ranks a node gets: a "universal rank" and a "strong rank". Every node gets the same universal rank (every node links to every other node with $1-d\times$ its weight; nodes with no outlinks are assumed to link to every node equally). We are first going to compute this universal rank:[3]
    - Within the iteration loop, iterate through the graph once and generate two sums: one for the ranks of all nodes with at least one outlink (r_0) and one for all nodes with no outlinks (r_1). Sums are computed from `oldranks`.
    - Compute the total universal rank as $r_0 \times (1-d) + r_1$. The value for d is 0.85 (given as the static variable `D` in the Java class).
    - Compute the share of universal rank for each node.
    - Initialise `newranks` with the universal rank for every node.
  - Next we compute the "strong rank", which is the rank that nodes give each other through links. Every node splits a ratio of d (i.e., 85% when $d = 0.85$) of its rank from the previous round (`oldranks`) equally with the nodes its links to.
    - Iterate through the graph. For each node with outlinks
      - Compute the share of rank it gives to its outlinks.
      - Iterate over the outlinks and add that share to each of their `newranks`.
  - Last but not least, we want to track some statistics from the computation. At the end of each iteration compute a sum of all ranks (`newranks`) and an epsilion (the sum of the absolute differences between `oldranks` and `newranks` for each node). Print both to console! Hopefully you see the sum stay close to one and the epsilon drop quite low.
  - One last thing ... any guesses what it is? ☺
- Run the code for both the test file and if the results add up to one, the full file (`-i` for input file, `-igz` if input is GZipped, `-o` for output file, `-ogz` to GZip output).
- Next run `SortByRank` for the output of the big file.
- Next call `OIDDecompress`. You need to pass as input the sorted rank file, the dictionary, as well as the column you want to decompress (`-n 0`).
- No need to submit yet. We continue on Wednesday.

[3] There's a couple of ways to do this **other than what's shown**. For example, knowing the sum is 1, we could just compute $\frac{1\times 0.15+0.85\times x}{n}$ where x is the sum of the ranks of all nodes with no outlinks.)