Information Retrieval: Assignment 2

Jolan Depreter
Lander De Roeck

November 2020

Contents

1 Introduction

Pageranking was the first algorithm used by Google to rank web pages. With the exponential growth of the web since then there is a need for more efficient implementations of the algorithm, to keep up with the ever expanding web. In this report we will look at possible ways to improve upon the basic pagerank algorithm. We will implement the mapreduce pageranking algorithm and will try to improve upon it. For the assignment we made use of the pyspark library and both the Google and ClueWeb data sets. The code discussed in this report is public on Github and can be found here: `https://github.com/jdepreter/ir-pagerank`.

2 Pageranking

Pageranking is a method developed to order the web pages on the Internet according to importance, it models the likeliness someone gets to a page through following random hyperlinks. It builds on the underlying logic that an important site will receive more references compared to other less important sites.

The algorithm is a link analysis algorithm, it assumes web pages as nodes and hyperlinks as edges in the graph. Pageranking can be implemented in two ways, algebraically and iteratively. The iterative approach is also called the power iterative method or just simply the power method. Since the algebraic method relies on the adjacency matrix, it is often preferred to use the iterative for big data sets, as the adjacency matrix grows quadratically.

The algorithm can be used to order search results from a query in a more reliable way, as the rank only depends on references on other sites. Which means site owners can't artificially increase their own rank on their site alone. They can create reference farms, the importance of the site however determines the weight of the reference, so doing this is pointless if the sites themselves have a low rank.

3 Graph Clustering

The original idea we had was trying to split the graph into "clustered" subgraphs. Then compute the pagerank in each of these subgraphs. Finally computing the pagerank between these subgraphs.

To cluster a webgraph we looked at several possible methods. One of the problems is that we don't know the amount of clusters present in the dataset. This means that something like k-nearest neighbour will not work. A very promising clustering algorithm is Density-Based Spatial Clustering of Applications with Noise (DBSCAN). There we are not required to specify the amount of clusters.

Our next problem is creating to correct input to start clustering. This proved to be quite the challenge. We either need a two dimensional representation of the graph, where each node is represented by a coordinate. Alternatively we

can provide a distance matrix. Since we will need a distance matrix to assign coordinates to nodes, we wanted to use a distance matrix.

Computing a distance matrix for such a large data set requires a lot of memory and computation power because we get a 50 million $\times$ 50 million matrix.

At this point, it became clear that trying to cluster the data would be almost as expensive as running pagerank without any optimizations. We also considered clustering after pagerank computation, but that would require a functioning and fast pagerank algorithm. So we switched our focus to that.

Clustering using pagerank has already been done successfully by other people [HST20]. This works as follows. Select k center nodes $c_i \mid i \in \{1, ..., k\}$ at random. Calculate the distance to every node from every center node c_i. Create subgraphs $G_i = (V_i, E_i) \subset G \mid i \in \{1, ..., k\}$ where V_i contains the nodes closest to c_i and $E_i \subseteq E$ the edges between nodes $v \in V_i$. Now calculate pagerank for each subgraph and select $v \in V_i \mid PR(v) = max_{w \in V_i}(PR(w))$ the node with the highest pagerank in the subgraph, as the new center node c_i. Do this for every subgraph and repeat by calculating the new distances, new subgraphs, new pageranks and new centers until the center nodes do not change.

4 Mapreduce pageranking

We started out implementing the algorithm that is detailed in the lecture materials, this implementation of a mapreduce pageranking algorithm will be used as a baseline against which we will compare our optimisation attempts. We will be mainly focusing on the execution speed, but also on things like CPU time and memory/storage requirements.

Algorithm 1: Map algorithm

```
1 Flatmap(key1: d_i, value1: (outlinks(d_i), PR(d_i))
2     for d_j in outlinks(d_i) do
3 |       (key2: d_j, value2: (1 − α)PR(d_i)/|outlinks(d_i)|)
4     end
```

Algorithm 2: Reduce algorithm

```
1 Reduce(key2: d_i, value2: PR_{1..m}(d_i))
2     PR(d_i) := (α/n) + Σ_{j=1..m} PR_j(d_i)
```

In the algorithm we assume the chance α, which corresponds to the chance to jump to a random node, and the total amount of nodes n. The algorithm will keep looping until the error is less than ϵ for every node.

4.1 Reading input

Both the datasets have another format, the goal is to create a sparse matrix in Spark. This matrix will be created as an RDD, which allows for mapreduce operations. For the clueweb dataset we create a RDD that contains all outgoing nodes as represented in the text file. Next we need the source node for these rows. This can be achieved with *rdd.zipWithIndex()*. However this is quite an expensive operation as it cannot be parallelized, taking up at least a minute and a half. To get rid of this unnecessary computation time, we can modify the text file by adding the source node before its outgoing transitions using some linux commands. For the Google dataset, which had an edge on each line, we create an rdd containing both the incoming and outgoing nodes. After which we use a *rdd.groupByKey()* to create an adjacency list.

The simplest MapReduce implementation of pagerank could not run on our hardware for the ClueWeb dataset. It takes a very long time to compute the *reduceByKey* and *join* operations. We were only able to measure the length of he first iteration: 37 minutes. Afterwards it crashed due to not having enough disk space.

4.2 Results

To keep consistency between the results, we used a single machine to produce all the results we used in the report. This machine used an Intel i7-4500U with 12GB of memory and running Ubuntu-18.04. This algorithm was only tested on the smaller dataset from google.

With the parameters: $\alpha = 0.15$ and $\epsilon = 10^{-6}$, this resulted in the values which can be seen in table 1.

# Iterations	**Error** ($\alpha = 0.15$)
00	0.0012783562990714412
01	0.0006503654092605307
02	0.0005528164763902806
03	0.00020954793813669163
04	0.0001531753262493103
05	0.00013020872666163265
...	...
30	2.239097129663677e-06
31	1.9032083582612121e-06
32	1.617623798950957e-06
33	1.3749630060204984e-06
34	1.168646786823982e-06
35	9.933376857865553e-07

Table 1: Iterations - error for each iteration

As we can see, we need 36 iterations before the maximum error becomes

smaller than ϵ. The total excecution time was 34 min and 30.3 sec, which is a rather long time for this "small" dataset.

4.3 Trying to optimize MapReduce

This was way too long for our liking so we started at which operations cost the most. The map and flatmap operations only take a few seconds each. The ReduceByKey operation takes up most of the time of an iteration. If we are able to make this operation smaller, we could possible lower the time.

Our first idea was to remove all nodes that do not have any outgoing nodes (sink nodes). We don't need to calculate their rank every iteration, since their rank is never used to compute another node's rank. We can calculate their ranks at the end. Unfortunately, to not calculate their ranks, we have to filter these nodes every iteration. It turns out that this is more costly than computing their ranks. The final time clocked in at 44 minutes and 57 seconds. This attempt is located at *./pyspark/google-graph/google-ranking-opt1.py*.

Our next attempt was to filter out nodes that are not visitable by any other node. The idea being that the score won't change in any iteration, so there is no point to keep computing these values. In a real context this would be sites that link to a lot of other sites, without ever being referenced themselves. Their score will always be equal to the teleportation chance. This did not add any extra operations to the iteration itself, and decreased the size of the ranking and links. It does however add some extra preprocessing calculations, to calculate these nodes and the ranks. While we expected this to perform very well, in practice it seemed to somehow be slower than the unmodified mapreduce algorithm, with the final time clocking in at 45 minutes and 50 seconds. This attempt is located at *./pyspark/google-graph/google-ranking-opt2.py*.

Both of our attempts didn't seem to actually improve the computation speed, hence we abandoned the map-reduce algorithm for now and decided to focus on other potential algorithms.

5 C++ Implementation

The main advantage of using C++ is that it is precompiled and generally is much faster. The downside being that it will have to keep a lot of data in memory, especially for bigger datasets, as it does not have any libraries like spark (to our knowledge). Keeping it all in memory should be much faster however.

We start by reading the input file. The webgraph is represented by a map with a node id as key and as value a vector of node ids, which correspond to the outgoing nodes. The node ranks will be kept in a separate second map. While reading the input file, we add every new unique node to this map and set its value to the base rank ($1.0/node\_count$). This representation is almost equivalent to the sparse matrix representation. The main differences being that

we do not store the degree for each node. We calculate it when needed. This reduces the memory overhead slightly.

Now we can start calculating the pagerank for each of the nodes. We start by creating a new map new_ranks that will contain the newly computed rank for each node. We can't edit the rank map directly because we need its old value to compute the new ranks. We loop over every key in the map and for each key we loop over its outgoing nodes. Each outgoing node's rank is increased by

$$(1-\alpha) \times PR(start\_node)/degree(start\_node)$$

When this loop is finished the new ranks for each node will be equal to

$$PR_{i+1}(node) = (1-\alpha) \times \sum_{s \in S(node)} \frac{PR_i(s)}{degree(s)}$$

with $S(node)$ = all nodes that link to *node*.
Then we only need to add the random teleportation probability to each node. This is a simple for loop over every key and adding $\frac{\alpha}{node\_count}$. Which finally gives us following formula.

$$PR(node) = \frac{\alpha}{node\_count} + (1-\alpha) \times \sum_{s \in S(node)} \frac{PR_i(s)}{degree(s)}$$

Finally we need to compute the max difference for a node between ranks and new_ranks to know when the ranking converges. This is a simple for loop.

5.1 Optimizations

When reading the big data set, we simply ignore nodes that do not have outgoing and incoming links. This is why, when running the C++ script, it will say there are only a total of 147.927.857 nodes instead of the expected 428.136.613. Since the removed nodes dont have any links associated with them, they will always score the lowest pagerank = $\frac{\alpha}{total\_nodes}$. This does influence the score given to our nodes, because our baserank is equal to $\frac{\alpha}{147.927.857} \neq \frac{\alpha}{428.136.613}$ but will not influence the relative rankings between nodes. This optimazation does not have any effect on the google dataset because they only provide nodes that have links associated with them.

5.2 Results

Iteration	Error ($\alpha = 0.15$)	Error ($\alpha = 0.10$)	Error ($\alpha = 0.50$)
0	0.001278360	0.001353550	0.000751974
1	0.000650365	0.000729129	0.000225040
2	0.000552816	0.000656223	0.000112521
7	9.40940e-05	0.000148645	1.34887e-06
8	7.99883e-05	0.000133795	6.74507e-07
34	1.16865e-06	8.63990e-06	
35	9.93338e-07	7.77582e-06	
54		1.04984e-06	
55		9.44847e-07	

Table 2: Errors for Google dataset using different values for α

We observe that when we lower the value of α, the amount of iterations needed for convergence increases (when using the same ϵ value of 10^{-6}). This can be attributed to the base teleportation chance (α) being lower, which means this base chance accounts for a smaller amount compared to the computed ranks of the incoming edges. Because only these incoming ranks are changing in between iterations, they solely contribute to the error. Since the weight assigned to these ranks is now increased, the error will respectively increase too. This is consistent with the fact that for $\alpha = 0.50$ we need less iterations to converge.

Let's take a look at the difference in pagerank. We take the top 10 nodes generated with $\alpha = 0.15$ and look at their new placement generated with $\alpha = 0.10$ and $\alpha = 0.50$

Node id	Rank ($\alpha = 0.15$)	Rank ($\alpha = 0.10$)	Rank ($\alpha = 0.50$)
597621	1	2	3
41909	2	1	7
163075	3	4	1
537039	4	3	2
384666	5	5	14
504140	6	6	8
486980	7	8	15
605856	8	11	4
32163	9	7	11
558791	10	10	13

Table 3: Rankings Google Dataset top 10 ($\alpha = 0.15$) compared to 0.10 and 0.50

Changing the value of α does change the placement of the nodes relative to each other. This could be because we do more iterations with a smaller α. So let's compute the pagerank with $\alpha = 0.10$ but limit the amount of iterations to 36, which equals the amount for $\alpha = 0.15$.

Node id	Rank ($\alpha = 0.15$)	Rank ($\alpha = 0.10$)	Rank ($\alpha = 0.10$), limited
597621	1	2	1
41909	2	1	2
163075	3	4	3
537039	4	3	4
384666	5	5	5
504140	6	6	6
486980	7	8	7
605856	8	11	8
32163	9	7	9
558791	10	10	10

Table 4: Rankings Google Dataset top 10 ($\alpha = 0.15$) compared to 0.10 and 0.10 with limited amount of iterations

The top ten is now equal to the top ten for $\alpha = 0.15$. We expected the positions for all nodes to be equal. However when we check the first 100, number 91 and 92 have switched places. This is probably because of a difference in rounding errors since the actual pagerank scores do differ and may cause floating point errors. We can conclude that α itself does not directly influence the ordering of the pagerank, α influences the amount of iterations, which in turn influences the ordering.

5.3 Resource Usage

As predicted, the C++ implementation takes up way more memory compared to Spark. Spark does not load all data into memory at once, it reads the parts of disk when it needs them. Our C++ implementation stores a representation of the file into memory. When the system has enough memory this means a tremendous decrease in computation time for the pagerank however (like for the google webgraph). To calculate the entire ClueWeb dataset, this implementation uses a staggering 21.5GB of memory (peak memory usage). Note that the C++ implementation only utilises a single core, so it can't scale with more cores or be modified to be run in a distributive way. The results generated by the C++ power iteration and the PySpark mapreduce implementation is equivalent.

α	PySpark	C++
0.15	34m 30.03s	7m 06.82s
0.10	1h 03m 07.46s	10m 53.36s
0.50	7m 36.86s	1m 57.64s

Table 5: Power Iteration Timings (Google webgraph)

6 Random walk interpretation

Instead of trying analytically compute the pagerank, we could also estimate by performing a simulation of a random walk. We start at each node and start walking randomly. The variable α will now not teleport us randomly but instead it will terminate our current walk. So we have $(1-\alpha)$ chance to go to one of the outgoing nodes. Of course if there aren't any outgoing nodes, we also terminate the walk. This algorithm is a Monte Carlo computation.

Convergence was defined by the maximum error between two iterations being less than ϵ. We define one iteration as a single random walk for all the nodes. We need at least one iteration before we can really start measuring errors. However for consistency the first error will compare this random walk iteration with the starting ranks which are:

$$PR_{start}(node) = 1.0/node\_count$$

After an iteration, the ranks computed in that random walk and the current ranks will be weighed and added. This will be our new pagerank.

$$freq_i(node) = \frac{visits_i(node)}{total\_visits_i}$$

$$PR_0(node) = freq_0(node)$$

$$PR_{i+1}(node) = \frac{PR_i(node) \times i + freq_{i+1}(node)}{i+1}$$

$visits_i(node)$ = amount of times the node is reached in all random walks for iteration i, this includes the one time the random walk starts at this node. $total\_visits_i$ is the total amount of visits for iteration i.

Because of the nature of Monte Carlo algorithms, the errors for each iteration will vary per run. This variation may cause the error to increase. However, we will always converge at some point, since the weight of $PR_i(node)$ will keep getting bigger and bigger. This causes the new frequencies to have less and less effect, which causes the error to keep going down.

The first error will always be the biggest. This is because the greatest error is caused by the pages with the highest page rank. Since these pageranks will already be estimated as being quite big, the difference between them and PR_{start} will be quite large.

Note that our reader is not different from the Power Iteration method. Thus we also remove nodes that don't have any links associated with them. We don't use the a "base rank" here, but the total_visits amount will differ. Because every node is supposed to be visited at least once, and we skip certain nodes. This will cause the total_visists to be smaller. However the same logic applies here. Nodes without links, and thus without rank, simply live at the bottom of the pagerank.

6.1 Advantages compared to Power Iteration

When comparing the top results of both the power iteration and random walk algorithms, which can be seen in table 6, we notice that the top results seem to overlap quite nicely. Both power iteration and random walk have very similar top tens after convergence. First and second place have switched places but if we compare their scores, they are very close to each other in both algorithms, with only a difference of only $2 * 10^{-6}$ and $4 * 10^{-6}$ respectively.

Node id	PI (C)	RW (C)	PI (1)	RW (1)	RW (1) run 2
597621	1	2	3	3	2
41909	2	1	21	1	1
163075	3	3	1	4	3
537039	4	4	2	2	4
384666	5	5	20	6	5
504140	6	6	10	5	6
486980	7	7	22	8	7
605856	8	8	4	10	8
32163	9	9	13	11	11
558791	10	10	14	7	9

Table 6: Rankings - PI = Power Iteration, RW = Randow Walk, (C) after convergence, (1) using only one iteration.

The main advantage of the random walk interpretation is that we can quite reasonably estimate pagerank in a single iteration. Of course, this is much faster compared to computing the pagerank until convergence. The top ten of 'RW (1)' and 'RW (1) run 2' stay relatively intact with only a few nodes shuffled compared to the others' top ten. While the first iteration of the power iteration algorithm has quite a few nodes missing from its top ten. Most notably, '41909', which ranks first in all random walks and second in the converged results of the power iteration method, but only ranks 21^{st} after a single iteration of the power algorithm. This makes running until convergence a requirement for the power iteration method.

Furthermore, the difference between successive iterations is way smaller for the first iterations of Random Walk. Further proving the point that a single iteration of Random Walk is already quite good as an estimation for the pagerank. This can be seen in table 7

Iteration	Power Iteration Error ($\alpha = 0.15$)	Random Walk Error ($\alpha = 0.15$)
0	0.001278360	0.000927567
1	0.000650365	2.29175e-05
2	0.000552816	1.1396e-05

Table 7: Error difference

An unexpected advantage of the random walk algorithm is that it is actually more accurate. When we calculate the sum of the ranks we should get a value of 1. However, when we calculate the sum for the power iteration, for both the PySpark and C++ implementation, we see that there are significant floating point errors. The sum is ≈ 0.69, calculating the sum for the random walk algorithm gives us $1.0001 \approx 1$.

This is because the random walk uses an integer to store the amount of times a node is visited. It only uses a floating point division at the end. One time to calculate the frequency and another to add an iteration's result to the previous iterations. In contrast, the power iteration uses one floating point division for every outgoing node for every node. Even after just one iteration of power walk our sum is already ≈ 0.86.

On bigger datasets, there is a clear time advantage for the random walk algorithm compared to the power iteration method.

	RW (C) $\alpha = 0.15$	PI (C) $\alpha = 0.15$
total time	39m 34,72s	1h 14m 20,78s
algo time	$\approx$ 9m	$\approx$ 44m

Table 8: Speed Random walk and Power Iteration Method (nodes 0 - 50 million)

Note that reading the inputfile and writing the output takes up the exactly the same amount of time for both algorithms and takes $\approx$ 30m. This is how we calculated "algo time". The writing speed of the outputfile can still be improved, but this left as an exercise to the reader.

6.2 Disadvantages

The algorithm makes use of randomness, this means that the results for different runs will always slightly differ. This is especially pronounced in small datasets, where only a few "walks" get executed. The highest ranking nodes will still have a small number of links pointing to them. A node can get a worse pagerank position even if only a single walk is terminated early or the walk decides to ignore the node a few times. In our testing we also noticed that an iteration of random walk takes approximately 10 seconds longer than an iteration in the power iteration implementation (with the Google dataset). However, this will vary depending on the dataset size and structure. Random walk will be faster in graphs that have short "paths", as that will force the "walk" to complete.

7 Program Usage

For simplicity, create a folder named "data" in the root of the git repo and add the datasets in that folder.

7.1 PySpark

Make sure you have the following installed: Spark, java. Also make sure you have the following python packages installed: pyspark, findspark

To run the big dataset (not recommended). First, prepare the dataset by adding line numbers and removing empty nodes.

```
awk 'BEGIN{i=-1} /.*/{printf "%d.%s\n",i,$0; i++}' ClueWeb09_WG_50m.graph-txt
> ClueWeb09_WG_50m_numbered.graph-txt
sed -r --in-place 's/[0-9]*\.$//g' ClueWeb09_WG_50m_numbered.graph-txt
sed -i '/^$/d' ClueWeb09_WG_50m_numbered.graph-txt
```

Then run the python script.

```
cd pyspark
python3 pagerank.py
```

variables like α, ϵ, input- and outputfiles can be changed by editing the variables at the top of pagerank.py.

To run the google dataset, run

```
cd pyspark/google-graph
python3 google-ranking.py
```

variables like α, ϵ, input- and outputfiles can be changed by editing the variables at the top of google-ranking.py. The two optimazations attemps are run in the same way. A new directory will be created containing the sorted csv file. Make sure this directory does not already exists, the program refuses to overwrite already written output. The output csv file is sorted on decreasing pagerank.

7.2 C++

The source code can be found in *./c++*. Compile main.cpp and run the binary with the following cli arguments

```
cd c++
g++ main.cpp
./a.out help     # gives cli explanation
./a.out inputfile format outputfile algorithm [alpha] [iterations]
```

inputfile: path to web graph file e.g. *../data/ClueWeb09_WG_50m.graph-txt*
format: graph-txt or google
outputfile: path to outputfile e.g. *out-power-0.15.csv*

algorithm: power or random (optional, default = power)
alpha: 0.0 - 1.0 (optional, default = 0.15)
iterations: max amount of iterations to run (optional, default = −1 (unlimited))

The csv file output of the C++ program is sorted on node id. To sort by pagerank use the following command on the outputted csv file.

```
sort --field-separator=';' --reverse --key=2,2 -g out-power-0.15.csv
> out-power-0.15-sorted.csv
```

8 Conclusion

If we only care about the top results, e.g. for ranking search results, a single iteration of random walk will be sufficient. However if there is a need to correctly rank nodes with few links associated to them or for smaller webgraphs, the power iteration method will be faster and more accurate. The values for α and ϵ will indirectly determine the accuracy of the positions of your pages. So choose them wisely.

References

[Das+15] Atish Das Sarma et al. "Fast distributed PageRank computation". In: *Theoretical Computer Science* 561 (Jan. 2015), pp. 113–121. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2014.04.003. URL: http://dx.doi.org/10.1016/j.tcs.2014.04.003.

[HST20] Mustafa Hajij, Eyad Said, and Robert Todd. *PageRank and The K-Means Clustering Algorithm*. 2020. arXiv: 2005.04774 [cs.LG].

A Appendix

A.1 Attached files

In the github release, you will find the pageranks computed with the C++ algorithms. We use the following naming scheme: dataset-algo-alpha(-max_iteration). For example: google-random-0.15-1.csv is the pagerank of the google dataset computed using one iteration of random walk and $\alpha = 0.15$

- Datasets
  - full: The complete ClueWeb dataset
  - partial: Nodes 0 - 50 million of the ClueWeb dataset
  - google: The google dataset
- Algorithms: power or random

A.2 C++ Algorithms

Algorithm 3: Power Iteration

```
1  int Iterate(ranks, links)
2      map<int32_t, double>new_ranks;
3      // Calculate chance to jump from page to page
4      for it = links.begin(); it != links.end(); it++ do
5      |   vector<int32_t>out = it->second;
6      |   for v_it = out.begin(); v_it != out.end(); v_it++ do
7      |   |   new_ranks[*v_it] += β * ranks[it->first] / out.size();
8      |   end
9      end
10     // Add random chance to jump to this page and Calculate max
           error
11     int error = 0;
12     for it = ranks.begin(); it != ranks.end(); it++ do
13     |   new_ranks[it->first] += base_rank;
14     |   error = max(error, abs(ranks[it->first] - new_ranks[it->first]));
15     end
16     // Update ranks to the new values
17     ranks = new_ranks;
18     return error;
```

Algorithm 4: Power Iteration Method

```
1  map<int32_t, double>ranks;
2  map<int32_t, vector<int32_t>>links;
3  readfile(file, ranks, links); // Initialize links, ranks
4  double error = 1;
5  while error < ϵ do
6  |   error = iterate(ranks, links);
7  end
```

Algorithm 5: Random walk iteration

```
1  random_walk(ranks, links, current_iteration)
2      int total_visits = 0;
3      map<int, double>visits;
4      for link in links do
5          bool first = true;
6          int current = link.source;
7          while true do
8              if !first then
9                  visits[current] += 1;
10                 total_visits += 1;
11             else
12                 first = false;
13             end
14             double r = random(); // [0.0, 1.0[
15             if r < α or current.degree == 0 then
16                 break;
17             else
18                 // Choose random node out of outgoing links
19                 r = (r - α) * 1.0 / (1 - α) * current.degree;
20                 current = links[current][(int)r];
21             end
22         end
23     end
24     total_visits += ranks.size();
25     double error = 0;
26     for rank in ranks do
27         // Compute distribution
28         visits[rank.key] = visits[rank.key]) / (double)total_visits;
29         if current_iteration > 0 then
30             visits[rank.key] = (rank.value * current_iteration +
                visits[rank.key]) / (double)(current_iteration + 1);
31         else
32         end
33         error = max(error, abs(visits[rank.key] - rank.value));
34     end
35     ranks = visits;
```