Project 3

Big Data

Name: Theodoros Mandilaras
A.M.: cs2.190018
MSc DIT/EKPA | Spring 2019-2020.

Page Ranking

The project implemented with **Python 3.7**. The given zip file should contain the below files:

- Page_ranking.py: It contains the main code of the implementation.
- Node.py: It contains the class that describes the nodes of the web graphs, as we are going to see.
- README.md: Contains information about the execution.
- Results: A directory with all the results.

Page Rank Class

The **PageRank** class is located in the page_ranking.py file. This class includes all the functionalities for this project. It creates the web-graph from the given text file, it implements the simple and the improved iteration algorithm in order to calculate the ranking, it generates the requested top20 and bottom20 results, it also creates the plots and lastly it contains the convergence condition for the last question of that project.

Node Class

The **Node** class is located in the node.py file. This class contains all the required variables and methods that are necessary for the correct and efficient execution of the page ranking. For each page in the web-graph text file, a node object is created. The members of this class are:

- Index: This member shows the position of the node in the ranking vector (and also in the matrix) which will be created. It starts from the 0 and ends in the *NumberOfNodes - 1*.
- Key: The key is the id of the node (page), which is the value that is given while the program parses the web-graph text file. This member is different with the index member, because the key is not always in the correct sequence (just like in the *web-Google.txt*).
- Edges: A list of node classed that contains those nodes that our node can lead to.
- Importance: The importance of the node. At the begging it starts with the value of 1. When the parsing of the web graph is completed, then the real value of the node is calculated in the **compute_importance()** method. The importance is calculated by the

division of 1 with the length of the edges list. However, if a node does not lead to any other node, then the importance stays as 1 because we cannot divide with zero, and also for those sites we want an increased importance.

- Vector: The vector member contains a csr (compressed sparse row) matrix from the scip.sparse module. Its shape is (1, *NumberOfNodes*) and the values in each position of this vector is the importance of the node in this position, if our node (*page*) can lead to that node (*page*) else the value is 0.

That was the description of the node class. Now we can see how the page ranking is done.

Page Rank Class Creating

To create the **PageRank** we have to provide two arguments. The first one is the path of the web-graph text file, and the second one is which algorithm to use (*simple or improved*). According to the given method, it assigns in the iteration() function the name of the correct method (simple_iteration() or improved_iteration()).

Graph and Nodeset

The graph is described with a dictionary which has as keys the page ids and as value the node object for that page. This dictionary firstly is initialized as empty. Moreover, a list called nodeset is created, in which all nodes will be added (once). Using that nodeset the program can extract the index value for each node (also the opposite) and in the end of the parsing, the total number of the nodes (length).

Creating the Graph

After that initialization, a csv reader is created and starts parsing the web graph file line by line. If a line is a comment, it is ignored (*lines starting with #*), else it extracts for each line the *page id* of the source and the *page id* destination. Firstly, the source page id is checked if it already exists in the graph. If it does not, it is appended in the nodeset list, and also it creates the **node** object for that page using the as *key* the page id and as *index* the correct position of that page in the nodeset(len(nodeset)-1 because it just was added). This node is added in the graph as value, and having for key its page id. Then, the same thing is done for the destination page. After that, in the node of the source page appends in the *edges* list the destination node.

When the parsing of the graph file is complete, the graph dictionary contains all the node. Also, the nodeset contains all the page ids, from which the program gets the number of the nodes (length).

The parsing of the graph file and the creation of the nodes graph takes **10.09 seconds**.

Calculating Importance

After parsing the graph file, we have collected all the edges between the nodes. So now, the program can safely calculate the importance of each node. The nodes are parsed in a for-loop in the graph dictionary and its importance is calculated with the compute_importance() method. This method calculates the quotient of 1/(number of the edges). If the node has no edges (by edges I mean the nodes that the node can lead to only) it gets 1 for importance. That happens because we cannot divide with zero and I believe that in those nodes we have to assign increased importance.

The importance part takes **0.34 seconds.**

Creating the Vectors

The M matrix of the web graph that it is described in the project is constant. In each row i in the column j, contains the importance of the node j if the node i leads to j, otherwise it is 0. Those rows are also constants. So, instead of creating the whole matrix, I create for each node in the graph its vector, which is the described row. This happens by collecting the importance of the edges and assign them in the correct positions (using the index member of the edge nodes) into a sparse (filled with zeros) vector with the shape of (1, *NumberOfNodes*).

The creation of the Vectors for all the nodes in the graph takes **163.5 seconds**.

Ranks

For initializing the ranks I create a numpy array with ones, with shape (*NumberOfNodes,1)*. In each position of that ranks array, will contain the rank score for the page with index value in the node, the position of the array.

Now the construction of the **PageRank** class is complete, and we can start iterating the desired algorithm.

Simple Iterate

The simple iterate is implemented in the simple_iterate() function and executes one iteration of the simple algorithm. In this function the program parses all (at least those that leads to others) nodes in the graph and calculate the rank score of that node. The rank score is calculated by multiplying the node vector with the rank array. That returns a number which is the score of the node in the current iteration. That score is assigned to the rank array in the correct position which refers to the current node, using the index member of the node.

Because, the rank array is updated in the process, while the program still parses the graph, we don’t want to use the updated rank array. So, the program copies the rank array before the

parsing starts, in the p variable, and uses that in the multiplication part which is not affected by the updates.

One simple iteration takes about **3.4 seconds**.

Improved Iterate

An iteration of the improved algorithm is implemented in the improved_iterate() function. The idea is almost the same with the simple iterate function. I calculate with the same way the rank array. However, after the program parse all the nodes in the graph, I multiply and add the values that are mentioned in the equation.

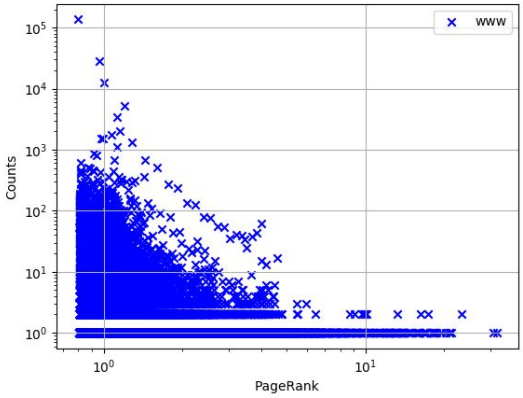
One improved iteration takes about **3.6 seconds**.

Creating the Histograms-Plots

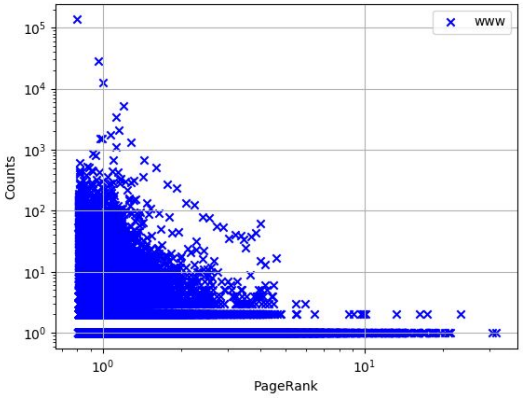
After executing the desired amount of iterations, in the rank array are stored the ranks scores for all the nodes in the graph. Then we can create the requested plot to visualize the results. The plots are created in the create_plot() method. For creating the plots I use the matplotlib module.

Approach 1

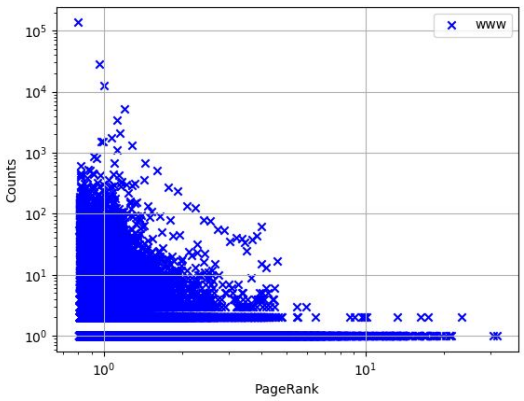
My first approach was to use the function of the numpy.unique() over the rank array. That function returns a tuple of arrays. The first array contains the rank values, and the second one contains the frequencies (counts) for those values. After that I create the plot normally by feeding it those two arrays. The results are the below:



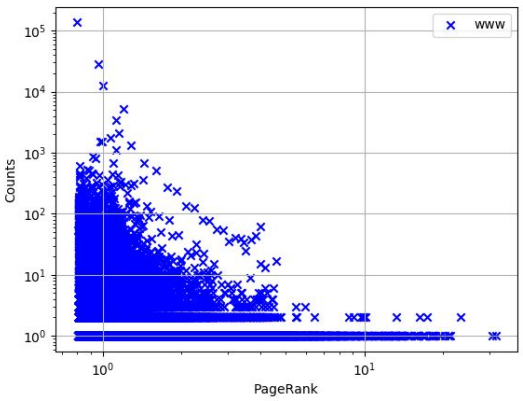
Improved with a=0.2 and 10 iterations



Improved with a=0.2 and 50 iterations



Improved with a=0.2 and 100 iterations



Improved with a=0.2 and 200 iterations

As we can see, the fact that the rank values are not very common, creates a lot of points (349.474 points) in the plot, making it non-readable. So I abandoned that approach.

Approach 2

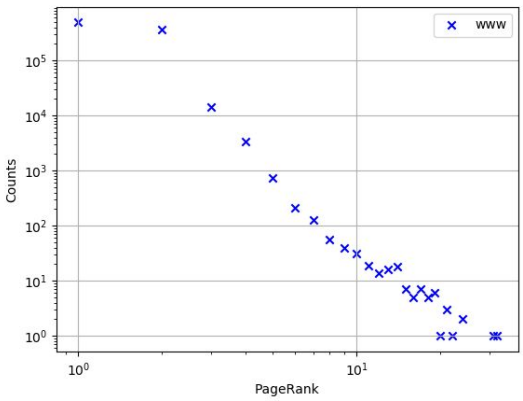
In my second approach I decided to round the rank results into integers, so by this way, they will not be so many different rank values, as results the counts will be increased and the number of points in the plot will dramatically decrease. In this way the plot can be readable. After rounding the values the counts (frequencies) of the values got using the numpy.unique().

Rounding with round()

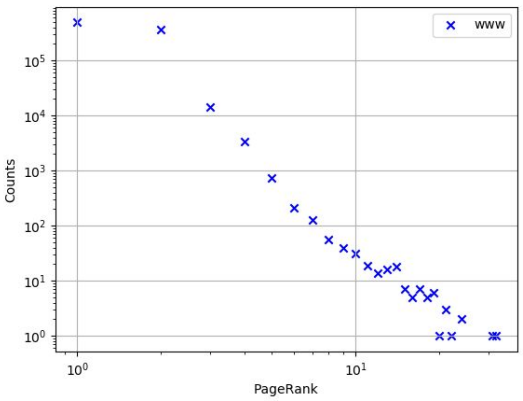
Firstly I rounded the ranks array with the numpy.round(). However, rank value near to zero will be rounded to zero, and log of zero is not defined. I could use the ***symlog*** scale for the axis but with this way the plots wound not be 100% same with the one that described (because they would contain the 0 lines).

Rounding with ceil()

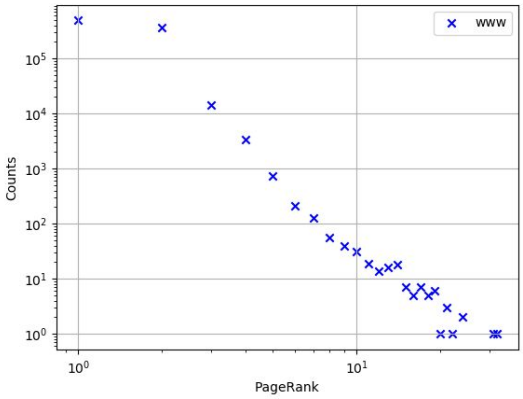
In order to avoid rounding to zero values, I round with the numpy.ceil() method. In this way the rank values close to zero will be rounded in the 1 and the logarithmic scale can be normally used in the axis. An example of that method can be seen below.



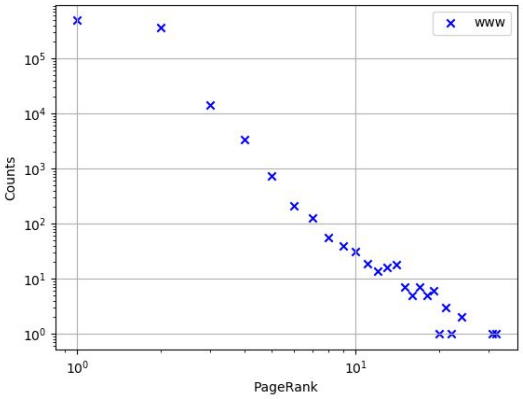
Improved with a=0.2 and 10 iterations



Improved with a=0.2 and 50 iterations



Improved with a=0.2 and 100 iterations



Improved with a=0.2 and 200 iterations

That is the method that I used to create the delivered plots. All the plots are located in the **results/plots** path in the .zip file.

Top/Bottom 20

The top or bottom 20 are found in the top() method. In this method you give the size of the top K values (20 in our case) and a weight. According to the weight it will return the top K or the bottom K. To calculate them, it uses the numpy.argsort() function, which returns the indexes of the values, starting from the smallest and going to the largest.
For example:

- To get the indexes for the bottom 20 I do:

```
ranks.argsort()[:20]
```

- To get the indexes for the top 20 I do:

```
(-ranks).argsort()[:20]
```

So the w can be 1 or -1. After that I can easily find the page ids with that ranks scores from the graph or the nodeset.

Those requested results are located in the **results/results** directory in the .zip file.

Graph Observations (**b question**)

With a first look on these histograms, we can see that most of the nodes(pages) are getting a low score, compared to very few others that are getting a great score.

Moreover, by looking all the plots for different iterations, we can see that as we increase the number of the iterations, the plots are not so different. The position of the points is not changing drastically, or it doesn't change at all. That means that the rank scores are **converging** into a value. Specifically:

- In the simple algorithm, the points in all the four plots (for 10, 50, 100 and 200 iterations) are changing. That means that, by 200 iterations the algorithm hasn't achieve to converge.
- In the improved algorithm, using ***a=0.2*** all the points in the plots are in the same place. That means that the rank scores have converged before the 10th iteration.
- Also, in the improved algorithms, using ***a=0.85***, the points in the plots stop moving after the plot of 50 iterations. That means the amount of iterations that needs to converge for that algorithm, is between 50 and 100 iterations.

Execution times (**c question**)

By executing the algorithm for the requested times, I am getting the below performances:

Iterations \ Algorithm	Simple (seconds)	Improved with **a=0.2** (seconds)	Improved with **a=0.85** (seconds)
10	210.2	214	214.6
50	351.05	354.9	356.4
100	533.2	532.7	536
200	882.59	894	896

All the execution measurements have been measure in the *web-Google.txt* graph.

Convergence (**d question**)

As I have already mentioned in the plot observation section, we can see that the position of the points stops changing as we increase the iterations. That means, that after a number of iterations, it is pointless to continue to iterate cause the rank scores are staying the same.

In order to find that point and avoid doing all those pointless iterations, we can stop the iterations when:

- The new rank array (p) after the iteration, is "*almost the same"* with the one before the iteration.
- After a max iteration limit (in case the ranks does not converge).

In my implementation, for checking the differences between the new and the last rank array (p), I subtract the members of those two arrays and I store the absolute values into a new array. Then, I sum all the members of this new array. If this sum is lower than a value then, the page ranking has converged and stops the iteration. I calculate that value as the product of the *NumberOfNodes* multiplied by a tiny number (in my case 1.96e-6). Source here.

So, after executing this method in our web-graph we see the below results:

- The simple algorithm could not converge even after 550 iterations (where I stopped it). Which is something that make sense, after seeing that the position of the points keeps changing in the plots, even after 200 iterations.
- The improved algorithm with ***a=0.2*** converges after *only 5* iterations, which agrees totally with all the plots that shows no changes in the points.
- The improved algorithm with ***a=0.85*** converges after 55 iterations, which also agrees with the plots, since they stop changing after 50 iterations.

Execution Modes

As I have mentioned and in the **README.md** file, the program can be executed in three different modes:

Default Mode

In the default mode, the program gets from the arguments the desired method, the path for the graph and the number of the iterations (if improved method has been selected, it also gets the numeric value "a"). Then, it performs the selected algorithm for the desired number of iterations. In the end, it stores the top and bottom 20 in a text file in the "results" directory, and also it makes the histogram for the calculated scores.

Evaluate Mode

The evaluate mode implemented to execute and store the outputs(*top20/bottom20 and the plots*) for all the requested iterations (10, 50, 100, 200) in a single run. By this way I could efficiently collect the results.

Converge Mode

In this mode, the selected algorithm iterates until it *converge*. That means that, it iterates until the sum of the absolute differences between last rank vector and the new one is less than a small value. When it does, it only prints the number of the iteration. So it does not store any results.

More information about the execution and the arguments exist in the README.md file, which has been delivered in the .zip.

Results structure

The program creates a directory called "**results**" in the *working directory*. Inside it, also creates three folders:

- **results**: In this folder the top and bottom 20 text files are stored. Files format: *<method>_<a value>_<iterations>_<bottom20 or top20>.txt*
- **plots**: In this folder the histogram plots are stored. Files format: *<method>_<a value>_<iterations>.png*
- **ranks**: That was a folder for development purpose. It is no more used.

THE END