**Exercise 2 – Middle Project**

**Degree Centrality**

The degree centrality algorithm is formed from a single cycle for which the degree of each node is divided by the number of nodes - 1.  
This algorithm is therefore extremely fast, taking about 0.2 seconds to complete the execution of the Naive version.  
Furthermore, a parallel version has been implemented by dividing the list of nodes into j parts, but, although there is no management of expansive parallelization due to the lack of data structures, this version was much slower than the Naive version and in particular it used about 10 seconds to be executed with 2 jobs and 20 seconds with 4 jobs.  
Given the speed and simplicity of the Naive version, it is not necessary to carry out other optimization operations.  
As far as the results are concerned, this algorithm trivially returns the 500 nodes with higher degree both in the Naive and parallel version.

**Closeness Centrality**

The closeness centrality algorithm performs the BFS (since the graph is not weighted and not directed) starting from each node. The calculation of this centrality measure is therefore very onerous, as it has been noted from the first experiments.  
The Naive version of this algorithm took just over an hour to run.  
Consequently, a parallel version has been implemented through the subdivision of the nodes into j parts to obtain better execution times. In addition, the parallelization still allowed to obtain the same results as the Naive version, proving to be an excellent solution.

With the use of 2 jobs, however, only an improvement of about 15 minutes was obtained, obtaining a running time of 50 minutes. With the use of 4 jobs, on the other hand, a clear speed up of the algorithm was obtained, which took 25 minutes to execute. No more than 4 jobs have been tested due to the physical limitations of the computer on which the algorithm was tested, but, in general, to obtain the shortest possible execution times, it is recommended to choose many jobs as the number of logical cores present on the computer.  
This solution was found to be satisfactory as it made it possible to obtain an execution time of 25 minutes, compared to 65 for the Naive version, without a loss of algorithm precision.

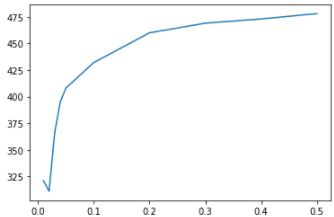
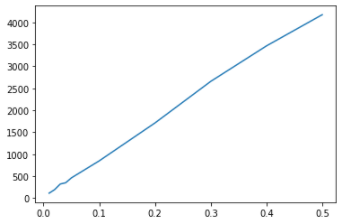
However, an attempt was made to improve this time by sampling the nodes on which to calculate the closeness. This sampling was done by extracting random nodes from the graph, excluding them from the set of nodes on which to perform the BFS, but including all its neighbors. This operation was carried out until each node had either been excluded or was the neighbor of an excluded one. This, therefore, allowed to obtain a measure of the centrality measure for all the neighbors of the exclusive nodes, thus being able to estimate the closeness of the excluded as an arithmetic mean of the closeness of its neighbors. This estimate, although causing a decrease in accuracy, was found to be suitable for this problem since for each calculated best path (u, v), considering the vertex x, excluding neighbor of u, the path (x, v) has a length between (u,v)+1 and (u,v)-1. We can therefore assume that using the average of the closeness of the neighbors for the calculation of the closeness of the excluded allows us not to deviate too much from the real centrality value.  
This version was performed in parallel with the use of 4 jobs, since, as previously mentioned, parallelization allows to obtain better execution times without loss of accuracy, thus obtaining a reduction in execution time of 10 minutes compared to the version without sampling, going from 25 minutes to 15 minutes. This operation allowed to exclude from the calculation of the BFS about 9000 nodes out of the initial 22470, obtaining however a result slightly different from that of the Naive. In fact, within the top 500, only 270 nodes are also present in the top 500 of the Naive version, where, in particular, only the element with the greatest closeness has seen its position remain unchanged. Furthermore, we can note that considering only the 10 elements with the greatest closeness given by the sampled version, only 6 nodes appear to be present in the real top 500. We can therefore conclude that considering the trade-off between accuracy and execution times, the use of sampling for this type of metric is not satisfactory. Consequently, for graphs of this size, it is preferable to use a simple parallel version.

**Betweenness Centrality**

The betweenness centrality algorithm computes all the short paths between each pair of nodes, which is extremely onerous.  
The Naive version of this algorithm took about 6 hours and 15 minutes to run.

Consequently, a parallel version has been implemented through the subdivision of the nodes into j parts to obtain better execution times. Furthermore, the parallelization still allowed us to obtain the same results as the Naive version, making it an excellent approach.  
With the use of 2 jobs, however, only an improvement of about 1 hour and 30 minutes was obtained, obtaining a running time of 4 hours and 40 minutes. With the use of 4 jobs, on the other hand, an adequate improvement was obtained, which took 2 hours and 20 minutes to execute. No more than 4 jobs have been tested due to the physical limitations of the computer on which the algorithm was tested, but, in general, to obtain the shortest possible execution times, it is recommended to choose many jobs as the number of logical cores present on the computer.

However, this execution time is still excessive, consequently, it was decided to carry out a random sampling to obtain acceptable times. This sampling proved to be an excellent solution, offering an excellent trade-off between execution times and accuracy. Two graphs are shown below which respectively contain the execution times and the number of nodes of the top500 also present in the top500 of the Naive version for each sampling factor tested (0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5).



**SCEGLIERE IL MIGLIOR TRADE OFF DELLA BETWEENNESS SAMPLED**

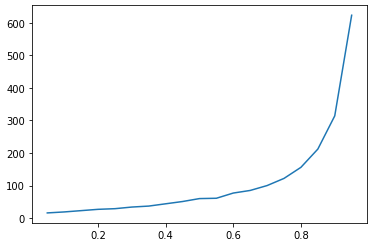
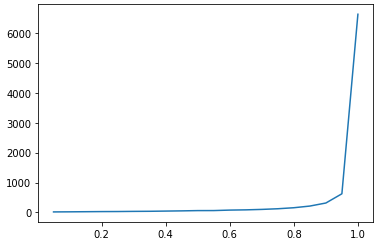
**Pagerank**

For the implementation of PageRank, the choice of the k and s parameters plays a role of primary importance, where k represents the number of rank updating cycles, while s represents the scaling factor, in which the value s\*rank (x) is divided between the neighbors of the node and the value (1-s)\*rank (x) between all the nodes of the graph.  
Note that, since the sum of all ranks is always equal to 1, the subdivision of (1-s)\*rank (x) among all nodes is equivalent to a starting rank equal to (1-s)/n , with n number of nodes, to which the fractions of the rank of the neighboring respects are subsequently added.

The algorithm was implemented in two versions, in the first the dictionaries were used to calculate the rank (dict version), while the second uses the product between matrix and vector (matrix version).  
The Update Rule in the dict version was applied according to the above reasoning, while in the matrix version we were faced with a problem. The matrix version provides k multiplications between the transpose of a particular matrix and the rank vector. This matrix is ​​constructed by obtaining the (sparse) adjacency matrix M of the stochastic graph of G, to which the formula s\*M+(1-s)/n is then applied, where the sum of (1-s)/n is applied to each element. This involves building a 22470x22470 size array with all elements different from 0 which causes too much memory usage. To solve this problem it was decided not to construct this matrix but to carry out the product of the transpose of the sparse matrix M with the rank vector to obtain a vector which is subsequently multiplied by s, to finally add (1-s)/n to each element of the new vector obtained.

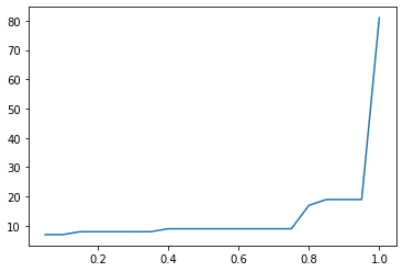
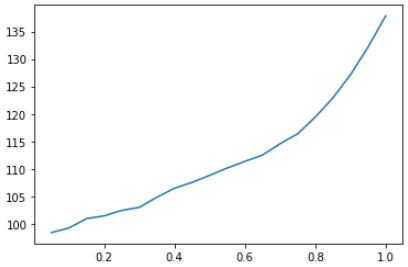
The versions were found to be equivalent in terms of cycles to achieve convergence, with only a slight difference between the results. In particular, the elements that make up the top500 remain unchanged, but there are small changes in positions. Such position changes are probably caused by the propagation of the error during the update cycles of the dict version. It was possible to verify that in the matrix version the sum of the ranks during each cycle is always equal to 1, while in the dict version this sum decreases as the algorithm progresses until a value of approximately 0.99 is obtained at its end. A total error of the order of 10-2 is very high considering that the average rank value is of the order of 10-5.

During the study of the s and k parameters, it was found that the number of k cycles required to obtain the convergence of the rank values ​​increases as s increases. For the study of convergence, the sum of the absolute values ​​of the differences between the ranks at the k-th and the (k-1)-th cycle was taken into consideration. In particular, only in a few cases, it has been possible to obtain a difference between consecutive cycles equal to 0, it is assumed due to machine errors, while in most cases this value decreases up to limit values ​​(or periodic patterns) of the order of 10-18 or less. The reaching of this limit was considered as convergence. Two graphs are shown below showing the number of update cycles k required to reach convergence for each scaling factor multiple of 0.05 between 0.05 and 1 in the first image and between 0.05 and 0.95 in the second (to best show the evolution for the first values ​​of s).



The graphs show how the relationship between k and s is of an exponential type, with a notable increase in the slope for values ​​greater than 0.9 and in particular for s = 1.

Subsequently, we moved on to the analysis of the results. For the various values ​​of s examined, 207 nodes were found present in each top500, even if in different positions. Furthermore, it was noted that higher values ​​of s tend to give a higher score to the nodes with a higher degree. Two graphs are shown below, in which the one on the left shows the average of the degrees of the nodes present at the top500 and the graph on the right shows the minimum degree of the nodes present at the top500.



In particular, it can be noted that for s = 1, the top500 is formed exactly by the 500 nodes with the highest degree since the graph is connected and not direct and consequently there are no traps.  
Furthermore, for values ​​of s<0.8 the presence of nodes of degree lower than 10 was found which allows us to conclude that low values ​​of s tend to give a high score even to nodes with few connections, while continuing to give a lot of importance also at high degree nodes, as shown by the graph of the average, almost always above 100.

The best parameter trade-off was then decided based on this information. In order to obtain short execution times with results that do not benefit too many nodes with few links, the value of s equal to 0.85 has been chosen as the best trade-off, which requires a minimum number of k cycles equal to 211 to converge. Furthermore, with this value, the average of the degrees of the nodes in the top500 turns out to be 122.922, while the minimum degree is 19. The value of k chosen to carry out the following experiments is 250, to be sure of reaching convergence, taking into account that an increase in such a small number of update cycles does not lead to major slowdowns.

Once the optimal values ​​of s and k (0.85 and 250) had been chosen, the study of the execution times of the various proposed algorithms was carried out. It should be noted that for the parallel implementations of the two versions a shared data structure was used between the jobs, a dictionary in the case of the dict version and a vector in the matrix version, given the need to know the ranks updated by other threads​​ in each thread and in each cycle. This choice made it possible to obtain results very close to those of the Naive versions, but not the same due to an imperfect synchronism between the jobs which can cause an imperfect updating of the data structure and therefore a reduction in the accuracy of the algorithm.

The Naive dict version takes about 5 minutes for the complete execution of the algorithm, an acceptable but not optimal time, as the Naive matrix version is much faster with an execution time of only 10 seconds. The related parallel versions, due to the very expensive lock management, do not allow to improve these times. The parallel dict version has an execution time of about 6 minutes with the use of 4 jobs and 5 minutes and 30 seconds with 2. The parallel matrix version, on the other hand, while still being very fast, involves a slowdown of a few seconds both with the use of 2 jobs and with 4.

The Naive matrix version is therefore both the fastest version, with very short execution times, and the most accurate as the parallel versions are unable to obtain the same results as this version, while the Naive dict version is affected by the problem of propagation of the calculation error.

**HITS**

The HITS algorithm is typically used on direct graphs and cyclically applies an Authority Update Rule (AUR) followed by a Hub Update Rule (HUR).  
Carrying out this algorithm on a non-direct graph leads to obtaining the same result for both the Hubs and the Authorities, as also found in the first experiments carried out. The AUR and the HUR are therefore completely equivalent, thus being able to obtain a generic Update Rule (UR) X=UR(X) such that two consecutive applications of UR produce the same result as the application of an AUR followed by a HUR.

The subsequent algorithms were therefore defined as the cyclic application of k UR which are equivalent to the application of k/2 cycles of AUR and HUR. Through various experiments, it was possible to confirm that this assumption is correct, obtaining the same results as output.

The algorithm was implemented in two versions, the first uses dictionaries for the calculation of the score (dict version), while the second uses the product between matrix and vector (matrix version).  
In both versions it was possible to notice a rapid convergence of the algorithm, as shown in the following images in which the ordinates represent the sum on each node of the absolute values ​​of the differences between the rank of the current cycle and the rank of the previous cycle.

|  |  |
| --- | --- |
| Dict version: | Matrix version: |
|  |  |

We can immediately see how the sum of the differences settles quickly around 0 already in the first ten cycles. The complete convergence is instead reached after about 150/200 iterations in both versions, with a settlement at 0 in the case of the dict version and a settlement at 3.5 \* 10-15 in the matrix version.

To obtain complete convergence and also given very short execution times, it has been chosen to set k equal to 200. In particular, the dict version takes about 63 seconds to perform 200 cycles, while the matrix version is much faster, completing the 200 cycles in about 2 seconds.

As for the parallel versions of these algorithms, the need to know the scores updated by the other threads in each thread and each cycle has forced the use of a shared data structure between the jobs. This choice therefore made it possible to obtain the same results as the Naive versions, both in terms of score and convergence, at the expense of speed due to the parallelization management times and the lock on the shared structure.

In particular, for the dict\_version, with 4 jobs the parallelization management becomes too expensive, taking about 65 seconds to complete the 200 cycles (about 2 seconds slower than Naive), while with only 2 jobs it was possible notice a moderate improvement in times, obtaining an execution time of 55 seconds.

On the contrary, for the matrix version, it was not possible to obtain any time improvement with parallelization due to both the very short time that the Naive version needs to run and the complexity of managing the parallelization.

In fact, with 4 jobs, an execution time of about 11 seconds is obtained, while with 2 jobs the time drops to about 7 seconds, in any case much higher than the 2 seconds that the Naive version takes on average.

It is possible to conclude that the best version is represented by the Naïve matrix version.

**Aggiungere confronto tra i risultati dei vari algoritmi**