UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica Applicata

Project of Social Network Analysis

July 2021

Group 13

Angellotto Daniele	0622701082
Cavalcante Marco	0622701209
Mignella Lorenzo	0622701269
Vigliotti Vincenzo	0622701206

A.A. 2020-2021

Summary

Exercise 1 – Middle Project
Hierarchical

The hierarchical clustering algorithm aggregates clusters, initially composed of a single node, with their own neighbour chosen arbitrarily, since each edge has the same priority.

The *naive* version of this algorithm has proven to be very expensive from the memory consumption side, as it keeps track of both edges and non-edges in a priority queue (i.e. those pairs of nodes that are not connected by a edge).
Let n be the number of nodes of the graph on which we are clustering, the algorithm inserts $n^2 - n$ elements in the priority queue, that is all the pairs made up of two different nodes. In the case of the graph we have considered, which is made up of 22470 nodes, the priority queue would contain about 504 million elements.
This heaviness is reflected not only in the amount of memory required, which makes the algorithm impossible to execute for the graph in exam, but also in the execution time of the algorithm: the search for nodes to fill the priority queue has a computational complexity of $O(n^2)$ and this queue must be iterated completely every time two clusters are aggregated, in order for the elements of the priority queue to be updated.
The version presented avoids this problem by considering only the edges, thus discarding the non-edge which are not useful for our purpose (not considering them forces us to perform one more search than the naive implementation, but being performed on a dictionary has negligible complexity), reducing the number of elements saved in memory and the complexity of the search necessary to fill the priority queue.

In detail, this modification brings the complexity of the search for the edges to be inserted to $O(m)$, where m is the number of the edges present in the graph, and the the number of edges saved in memory to m.
The behaviour remains almost identical: taking the first edge from the priority queue, the two clusters obtained are merged in a single cluster and the other edges, in which one of the two clusters obtained at the beginning was present, are updated, replacing them with the union of the two. Each insert in the priority queue is done in a way that is given the highest priority to the smallest clusters. This precaution was taken in order to avoid to create a single large cluster, in fact inserting all the edges with the same priority, the lasting updated ones would end up being the first to be extracted at the next iteration, thus leading to the creation of a single large cluster).
The execution of the algorithm took 309.28 seconds. The results obtained are shown in the following table, in which it can be observed how many nodes of each of the obtained clusters belong to the real clusters of the graph.

	Cluster1	Cluster2	Cluster3	Cluster4
Politician	1420	1263	2039	1046
Government	1820	1365	2399	1296
TVshow	967	786	935	369
Company	1748	1551	1942	1254

We can notice that the nodes that belong to the real cluster are not recognized from the hierarchical clustering, so they tend to be present in every cluster. The resulting accuracy is equal to 5976 out of 22470.

In conclusion, this implementation does not lose precision compared to the naive one, as it maintains the key elements of the initial algorithm, but it greatly lighten the workload and makes it possible to obtain a result in which the nodes are not part of a single cluster, but they are equally distributed.

K-Means

Two aspects play a role of primary importance for the implementation of k-means: the choice of the initial centroids and the definition of the distance between nodes and clusters.

Given the need to find (at least) 4 clusters within the graph, the initial step consists of choosing 4 centroids. These centroids, in each execution of the algorithm, are chosen randomly through a procedure that ensures that they have no connections to each other.
As far as distance is concerned, it was decided to define the cluster containing the greatest number of neighbors of that node as a cluster with minimum distance from a node.

Once these aspects have been defined, we moved on to the implementation of the algorithm, in which a fifth "cluster" was defined, without a centroid, populated by all the nodes that cannot be inserted into the other groups. In the case of an unconnected graph, it is populated by all the nodes of the components that do not contain a centroid, while in the graph under examination (which is connected), it is of fundamental importance for correct implementation of the parallel version, discussed in more detail below.

The Naive version of this algorithm (k_means function in the kmeans.py file) was found to be unacceptable for practical purposes given the too long execution time. In fact, the algorithm, which was tested on the Google Colaboratory online server, classified only 5000 elements out of 22470 in 6 hours (the execution was subsequently stopped manually), also showing an ever-increasing slowdown in the classification of the elements. Furthermore, the partial results are unsatisfactory given the tendency to overpopulate some clusters and leave others almost empty, as shown here.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Politician	289	0	1135	0
Company	289	18	528	0
Government	217	0	1974	2
TV-show	80	0	468	0

This experiment therefore highlighted two major problems to be solved: the execution time and the imbalance of the clusters.

First of all, it is of greater importance to reduce the execution time as much as possible to speed up the analysis of the problem. A parallel version of the algorithm was then implemented (parallel_k_means function in kmeans.py) and the related experiments were performed with 4 jobs.

The first problem encountered in this version was the inability to assign all nodes to clusters with a single execution of the algorithm. In fact, by dividing the nodes into 4 groups, there is a very high probability of having nodes not connected with the other nodes of its group. Consequently, as mentioned above, a fifth cluster was inserted that contains all the non-assignable nodes within its group and it was decided to make the parallel call to the clustering function iteratively, classifying the nodes in each cycle remaining from the previous run.

In the case in which the number of elements not yet assigned is less than 10*j, in which j represents the number of jobs, it is preferable not to execute the algorithm in parallel but in the naive version, since with a low number of elements the parallelization it has no advantages.

The parallel version was found to be much faster than the naive one, however requiring an unacceptable time, of about 8 hours and 30 minutes, with the execution of 4 clustering cycles, in which after the first 9118 elements remained unclassified, after the second 1251 and after the third 5. The results are shown below.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Politician	3851	410	1108	399
Company	3994	317	1295	889
Government	4748	608	977	547
TV-show	1995	146	744	442

You can immediately see how the accuracy is very low, failing to correctly identify the clusters. Furthermore, although it seems to be less affected by the problem of the imbalance of cluster sizes than the naive one, this problem recurs identically, as it was noted that, in the first cycle, each job tends to populate almost only one cluster, but they are likely to populate different clusters.
Before trying to solve this problem, it was however decided to work to further reduce execution times.

From an analysis of the algorithm, it was assumed that the main bottleneck is given by the instruction used to derive the list containing the nodes not yet added in a cluster that have at least one neighbour in them, which has complexity:

O (n*(sum of cluster lengths)*4*min(len(G.neighbours(el)),len(cluster)).

To eliminate this bottleneck, it was decided to remove the nodes from the samples variable when they are assigned to a cluster, to progressively reduce its size, and to introduce a new set: neighbours clusters. This set contains all the nodes already classified and their neighbours. So, whenever a node is added to a cluster, that node is inserted into the set and the union between the neighbourhood of the node and the set is made.

The new versions of the algorithm (optimized_k_means and parallel_opt_k_means) are extremely faster, with equivalent results in terms of accuracy and imbalance in cluster sizes. In particular, the normal version takes about 1 minute and 30 seconds to run, while the parallel version takes about 5 minutes. Note that the parallel version is much slower as it has to be run multiple times to assign all items to a cluster. The results of these versions have not been added to the documentation as they are equivalent to the previous ones and do not introduce possible new considerations.
Having reached such a short execution time, it is now possible to try to solve the problem of the imbalance of the dimensions between clusters and improve the accuracy of the algorithm through operations that, while slowing down the algorithm, allow to reach a better trade-off between execution time and accuracy.

It can be noted that a random choice of the centroids can cause particular situations in which the degree of a centroid is much greater than the others, obtaining a much greater probability of assigning an element to that cluster than the others. Considering also that there are 2658 nodes with degree 1 and at the same time various nodes with degree>200, this problem cannot be ignored.

To overcome this problem, it was decided to take as centroids the nodes of the highest degree that do not have connections between them. Furthermore, to try not to unbalance too much the probability of inserting elements in each cluster already at the beginning of the algorithm, it was decided to insert the elements present in a priority queue, whose priority of the elements is equal to their degree, thus giving higher priority to lower grade elements. This version achieved the same speed as the previous version, about 1 and a half minutes. Furthermore, with this implementation, all the random component of the algorithm has been removed, thus obtaining the same result in each execution, the result reported here.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Politician	96	1790	3800	82
Company	103	574	5811	7
Government	960	1015	4512	393
TV-show	304	416	2602	5

We can immediately see how the imbalance of the dimensions is still present but in a more reduced way than the previous non-parallel versions. As far as accuracy is concerned, it is still very low despite some slight improvements, correctly clustering 8566 elements out of 22470.

As for the parallel implementation of this version, even in this case the execution times remained approximately unchanged, taking about 5 minutes. The results are shown below.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Politician	933	1654	2972	209
Company	2186	159	4098	52
Government	1776	1030	3467	607
TV-show	1043	181	2088	15

Also, in this case we can see how the assignment of real clusters to the clusters obtained remains equivalent. From an initial analysis of the results, it can also be seen that TV-shows are more difficult to cluster through k-means than other categories. Also, the performance is lower than that of the non-parallel version, correctly assigning 7543 elements out of 22470 to clusters.

Girman-Newmann

The Girman-Newmann clustering is based on the edge betweenness measure and it iteratively removes the edges with highest betweenness until a stopping condition is met.

The naïve implementation of Girman-Newmann computes the edge betweenness for every node from every node in the graph once at time and then proceeds to remove the highest betweenness edge. The betweenness computation has a computational complexity of $O(nm)$, where n and m are respectively the number of nodes and edges inside the graph.
The algorithm stops once it has found 4 connected components. This condition was chosen in place of a condition based on the clustering performance because the latter takes way more time on the graph we took in consideration, since as shown in the table below we can see how removing edges tends to create very small connected components.
The execution of this implementation took 26215 seconds (about 7 hours).

	Cluster1	Cluster2	Cluster3	Cluster4
Politician	5748	20	0	0
Government	6880	0	0	0
TVshow	3327	0	0	0
Company	6480	0	14	1

You can see how clustering produces excessively unbalanced clusters, also having to associate a cluster in which they are not present to TVshows, obtaining a correct classification of 6914 samples out of 22470. Furthermore, the execution times are excessive and unacceptable.

Consequently, a parallel calculation of the betweenness was carried out, which turns out to be the most computationally onerous operation. This parallelization did not introduce approximations concerning the Naive version, thus obtaining the same results reported above. The running time was found to be 9146 seconds (about 2 hours and 30 minutes), still too high to be acceptable.

As also found in the second exercise, a high sampling for the calculation of the node betweenness allows to obtain very accurate results to the naive. This property was also preserved for the edge betweenness,

albeit to a lesser extent. In particular, with a sampling of 5%, considering the 500 edges with the highest betweenness, 153 of them are also present in the top500 of the Naive version, where however the edges with the highest betweenness are the same as the Naïve version. So, the optimized implementation computes the betweenness from a subset of nodes, obtained by a random sampling and divides the computation in parallel threads. In this case the computational complexity drops significantly, since now we have $O(n'm)$, where $n' = \frac{\mathrm{rn}}{\mathrm{t}}$, r is the percentage of nodes considered and t the number of threads in execution.
In the presented solution we used 5% of the total nodes and divided the computation into 4 parallel threads.
The introduction of a sampling with 0.05 ratio to 473 seconds (about 8 minutes). In the table below we show the results with sampling and parallelization.

	Cluster1	Cluster2	Cluster3	Cluster4
Politician	5746	2	20	0
Government	6839	41	0	0
TVshow	3327	0	0	0
Company	6480	0	0	14

The optimization introduced drastically reduced the computational complexity and slightly the accuracy, making this trade-off acceptable.

Spectral

The spectral clustering algorithm allows to find two clusters based on the sign of the eigenvector associated with the smaller eigenvalue, but having to associate the nodes of the graph to at least four groups there are two possible ways of operating: taking into consideration the eigenvectors associated with the two minor eigenvalues or perform the calculation a second time on the first two clusters obtained.

From the first observations, we can immediately see that the calculation of the radio eigenvalues and eigenvalues on matrices of such dimensions is both temporally and spatially very onerous, thus deciding to use the first two eigenvectors to reduce the execution times.

The naive algorithm has shown, like the other clustering algorithms, the problem of imbalance between clusters, an index of a non-clear separation into groups of the nodes and a very high probability that in this graph there is a giant component that contains almost all the knots. Moreover, it was found to be extremely slow due to the too onerous calculation of the eigenvectors with an execution time of about 13h and 30min. The results are reported here.

	Cluster1	Cluster2	Cluster3	Cluster4
Politician	0	1	0	5767
Government	0	0	0	6880
TVshow	49	6	212	3060
Company	11	115	126	6243

We can immediately notice that both the groups relating to 'Politicians' and 'Government' are completely contained in cluster4 (except one element), thus having to associate the former with a cluster in which their elements are not present to obtain accuracy. as much as possible. In particular, the best possible association between clusters and groups makes it possible to obtain 7207 correctly assigned nodes out of 22470

First, we try to reduce the execution time as much as possible. Since it is not possible to calculate the eigenvalues in a parallel manner, a light sampling of the nodes is performed by excluding from the graph and the eigenvalue calculation all nodes of degree 1. This decision was made based on two main factors: by removing all the nodes with a degree equal to 1 from a connected graph, the graph certainly remains connected; these nodes can only belong to the cluster in which their only neighbor is present, thus being able to assign them a cluster at the end of the algorithm.

Despite being a very small sampling, this allows us to reduce the nodes by just over a tenth, from 22470 to 19812, and, considering the computational complexity equal to $O(n^3)$, we obtain $O((9*n/10)^3)=O(729*n^3/1000)$. We, therefore, expect to reduce execution times by at least one-fifth.
We can see how the algorithm, with this sampling, takes 10h to be fully executed. It is, therefore, necessary to make other improvements to obtain acceptable times.
Below are the results of the algorithm with node sampling.

	Cluster1	Cluster2	Cluster3	Cluster4
Politician	1	0	5767	0
Government	0	0	6866	14
TVshow	86	0	3241	0
Company	140	40	6282	33

From the results, however, we can immediately see how on this occasion, in addition to the 'Politicians', also the 'TVshows' appear to have to be associated with clusters that do not contain their elements to maximize accuracy. Specifically, we got 7006 successfully paired nodes out of 22470.

To reduce the execution time it was therefore decided to use a different algorithm for the computation of the eigenvectors since the Arnoldi method used by scipy is too expensive for the computation of n-1 eigenvectors and not accurate if you want to calculate a lower number of eigenvectors.
It was therefore decided to use the inverse power method which calculates an approximation of only the dominant eigenvector of the inverse matrix through k products between matrix and vector without the need to find the relative eigenvalue. Note that the dominant eigenvector, the eigenvector relating to the eigenvalue with maximum value, of the inverse matrix corresponds to the eigenvector relating to the minimum eigenvalue of the original matrix.
However, this method allows to calculate of only one eigenvector and consequently, to divide the graph into four clusters, once the first result is obtained, further executions of the inverse power method are carried out. This method is rerun on both clusters obtained if the cluster with a smaller size contains at least one-tenth of the total elements, otherwise, it is rerun twice on clusters with the larger size. In the case in question, the difference in size between clusters is excessive and therefore the second solution is applied.
Even in this method, however, there appears to be a problematic bottleneck represented by the calculation of the inverse. In fact, in addition to requiring very long execution times, in the various experiments, this operation led to complete saturation of the ram after an hour of execution and, given the need to perform this calculation 3 times during the algorithm, this results not acceptable.
Further approximations were then used to calculate the inverse. The method that was most suitable for this purpose was the LU-decomposition which allows us to find an approximation of the inverse in a much shorter time.

Once at this point, various experiments were carried out to choose the number of 'k' update cycles of the inverse power method. It was thus concluded that after many cycles equal to the number of nodes in the graph, the clusters obtained by this method no longer change as the cycles increase. We, therefore, set k=n.

This version of the spectral algorithm was found to be much faster with a runtime of about 15 minutes. The results are shown below.

	Cluster1	Cluster2	Cluster3	Cluster4
Politician	0	0	5731	37
Government	1	2	6836	41
TVshow	0	0	3304	23
Company	2	5	6463	25

Like the previous results, also in this case we have that a cluster contains almost all the nodes with a further reduction in the size of the other clusters and the nodes correctly associated with the clusters have dropped to 6878 out of 22470.

Despite the excellent execution times, an experiment was also carried out on the sampled version of this algorithm. In this case, however, the execution times did not differ much, with a reduced execution time of only 30 seconds, while in terms of accuracy, clusters were obtained even more unbalanced in size compared to the latter version.

Exercise 2 – Middle Project

In this section, there is the description and the discussion of the algorithms used for the calculation of the various centrality measures.
Note that all the parallel versions are tested with a maximum of 4 jobs due to the physical limitations of the computer on which the algorithms were 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.

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 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.

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.
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.

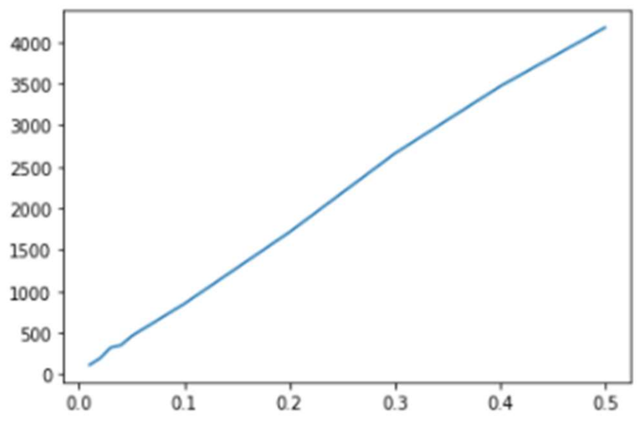
However, an attempt was made to improve this time by sampling the nodes on which to calculate the closeness. This sampling was done by extracting random nodes from the graph, excluding them from the set of nodes on which to perform the BFS, but including all its neighbours. This operation was carried out until each node had either been excluded or was the neighbour of an excluded one. This, therefore, allowed to obtain a measure of the centrality measure for all the neighbours of the exclusive nodes, thus being able to estimate the closeness of the excluded as an arithmetic mean of the closeness of its neighbours.
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 top500 of the Naive version. Furthermore, 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.
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.

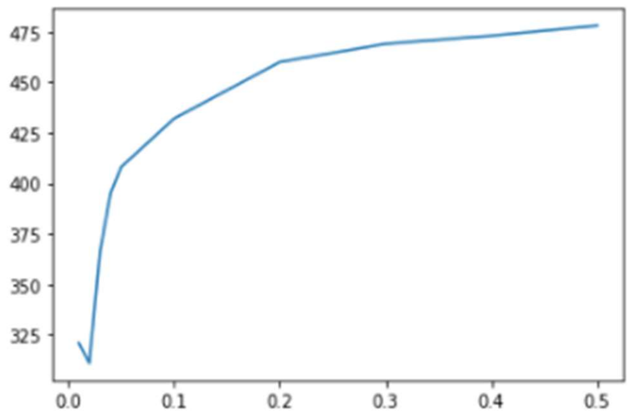
Betweenness Centrality

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.

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 graphics 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).





In order to obtain not excessive execution times, the best trade-off is given by the sampling with a factor of 5%. This sampling, even if it excluded a lot of nodes, produces a top500 very similar to the result of the Naïve version, with 408 common nodes, and an execution time of 8 minutes. Among these 408 nodes, only 8 are in the same position respect the Naïve version, but in the most of the case there are little change in the nodes positions.

Pagerank

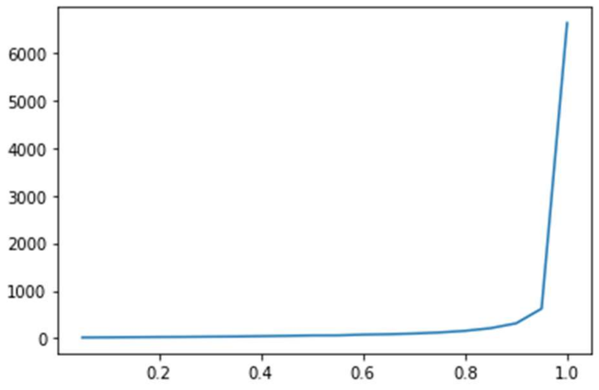
For the implementation of PageRank, the choice of the k and s parameters plays a role of primary importance, where k represents the number of rank updating cycles, while s represents the scaling factor, in which the value s*rank (x) is divided between the neighbours of the node and the value (1-s)*rank (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 neighbouring respects are subsequently added.

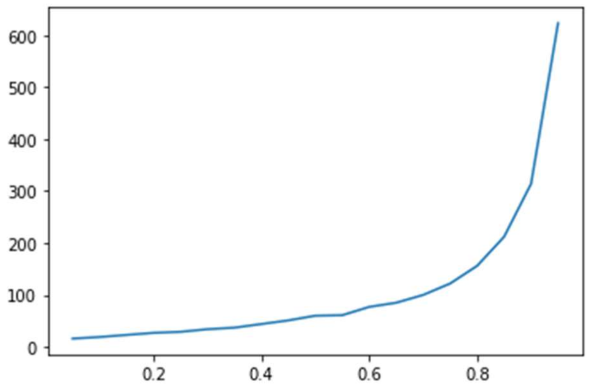
The algorithm was implemented in two versions, in the first the dictionaries were used to calculate the rank (dict version), while the second uses the product between matrix and vector (matrix version).
The Update Rule in the dict version was applied according to the above reasoning, while in the matrix version was chosen a different approach due to 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.

These two 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}.

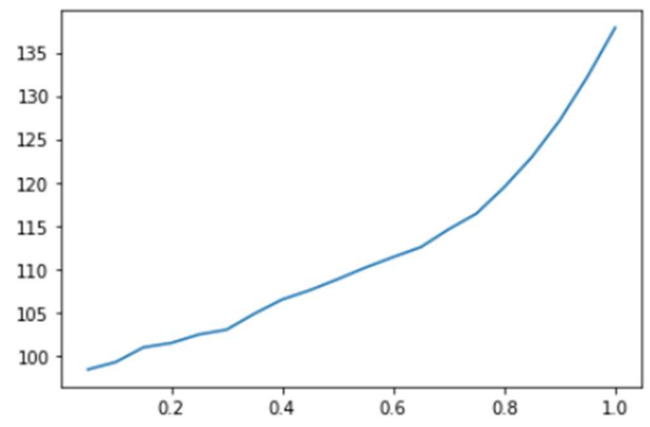
During the study of the s and k parameters, it was found that the number of k cycles required to obtain the convergence of the rank values increases as s increases. For the study of convergence, the sum of the absolute values of the differences between the ranks at the k-th and the (k-1)-th cycle was taken into consideration. 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 graphics 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).

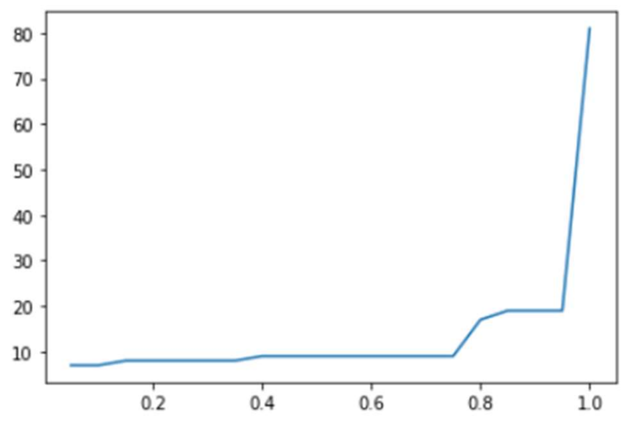




The graphics show how the relationship between k and s is of an exponential type, with a notable increase in the slope for values greater than 0.9 and, in particular, for s = 1.

Subsequently, there was the phase of 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 graphics 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 graphic 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, considering 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

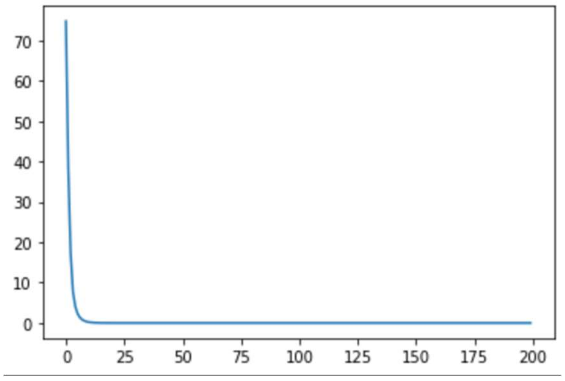
The HITS algorithm is typically used on direct graphs and cyclically applies an Authority Update Rule (AUR) followed by a Hub Update Rule (HUR).
Carrying out this algorithm on a non-direct graph leads to obtaining the same result for both the Hubs and the Authorities, as also noted in the first experiments. 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.

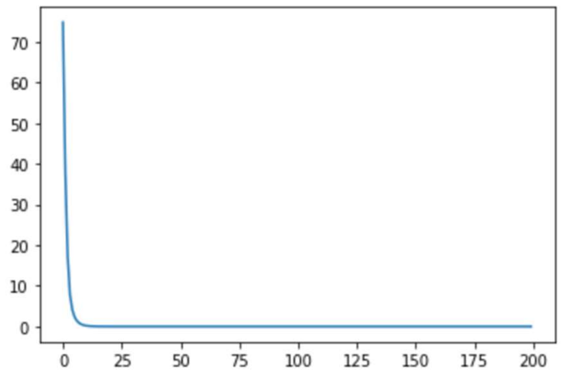
The subsequent algorithms were therefore defined as the cyclic application of k UR which are equivalent to the application of k/2 cycles of AUR and HUR. Through various experiments, it was possible to confirm that this assumption is correct, obtaining the same results as output.

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 represents 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:



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 2 seconds.

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.

Similarities and the differences among the result

In this section are considered only the result of the best versions of each algorithm, chosen at the end of each subsection.
In the next table are reported the number of common nodes between the various centrality measures.

	Degree	Closeness	Betweenness	Pagerank	HITS
Degree	---	235	185	323	218
Closeness	235	---	185	192	214
Betweenness	185	185	---	262	72
Pagerank	323	192	262	---	122
HITS	218	214	72	122	---

It can be noted that the measure of centrality most similar to the others is the Degree centrality. It has the greatest number of common nodes with Closeness, Pagerank and HITS. Only the Betweenness turns out to be quite different from the others, with a high similarity only with Pagerank. This similarity between Pagerank(s=0.85) and Betweenness and considering that the Pageranks with s = 1 return the same nodes as the Degree centrality shows how the Pagerank tends to give a lot of importance to nodes with a high degree as s increases and, at the same time, it tends to give greater importance to nodes with high betweenness as s decrease. From the results, it can be also noted that HITS gives importance to the node with a high degree and closeness, but it tends to exclude from the top500 the nodes with high Betweenness and Pagerank. Closeness is the only measure that has on average the same number of nodes in common in the top500 for any other measure.

Finally, the intersection between all the top500 shows that there are 60 nodes present in all the top500.

Exercise 3 – Middle Project

In order to provide a classifier that respects the assigned specifications, an algorithm has been implemented which, using three consecutive min cuts, classifies the samples from 1 to 3 stars.

First of all, it was necessary to generate the ground truth: all 288 possible combinations of acceptable votes were considered, for each of them the average of the votes was calculated (each feature has the same weight in the calculation of the average). Three bands have been established to assign the score:

- $mean < 2$ → 1 star
- $2 \leq mean < 4$ → 2 stars
- $mean \geq 4$ → 3 stars

The ground truth must not be interpreted as an assignment table, but as an indication for the critics: each of them, in fact, has the freedom to deviate from those indications, especially in extreme situations, i.e. when the mean of the score is close at one of the two thresholds.

For the generation of the first dataset, the deviation of the mean from the value of 3 is considered, calculating a probability as follows:

- 0, if $mean = 0$
- ½, if $mean = 3$
- 1, if $mean = 5$
- Values between 0 and ½, if $mean < 3$
- Values between ½ and 1, if $mean \geq 3$

For the generation of the second dataset, the deviation of the mean from the value of 2 is considered, calculating a probability as follows:

- 0, if $mean = 0$
- ½, if $mean = 2$
- 1, if $mean = 5$
- Values between 0 and ½, if $mean < 2$
- Values between ½ and 1, if $mean \geq 2$

For the generation of the second dataset, the deviation of the mean from the value of 4 is considered, calculating a probability as follows:

- 0, if $mean \leq 2$
- ½, if $mean = 4$
- 1, if $mean = 5$
- Values between 0 and ½, if $2 < mean < 4$
- Values between ½ and 1, if $mean \geq 4$

The probabilities were thus chosen to reflect the ground truth: in fact, a sample with a small mean has a very low probability of receiving a high score, and vice versa.

The three datasets were therefore generated:

- for each element its probability is calculated as described above
- n samples are generated (in particular for each sample, a random number between 100 and 500 is chosen)
- for each of them, a label 0 and 1 is randomly assigned, considering the probability assigned to it.

The distributions are calculated starting from these three datasets, considering the number of positive and negative labels for each sample: the distribution $D^+(x)$ is equal to the number of positive samples over the total number of samples, and $D^-(x) = 1 – D^+(x)$.

The first min cut is performed on the first distribution. It creates two partitions of the sample space, and the min cut is rerun on each of them. In particular, on the first partition it is re-executed with the second distribution, thus obtaining two further partitions: in the first there will be samples with 1 star, in the second one part of those with 2 stars. On the second partition it is re-executed, however, with the third distribution, obtaining also in this case two further partitions: in the first there will be the remaining part of the samples with 2 stars, in the second one those with 3 stars.

In order to classify a new sample, it has just to be check in which of the resulting partitions it appears to get the corresponding score.

Finally, the verification of truthfulness was carried out. All samples with three features and a score of 1 and 2 stars have been classified and for each of them are generated both of the possible samples with a ‘*’, in order to check that their score is not higher than the original sample. The result is positive.

Exercise 4 – Middle Project

Two alternative solutions were examined for the resolution of this exercise: Hill Climbing and Incentive Compatible Logistic Regression.

Hill climbing uses two classifiers (in our case we have chosen two linear regressors) trained on different spaces of features: in particular, the first is trained on samples with only 2 features (i.e. those samples that have a '*' in one of the two features in which is allowed, i.e. service and value), while the second on those with all 3 features. When a new sample is classified, if it has a '*' in one of the two features in which it is allowed, only the first regressor is used. Instead, if it does not have a '*':

- The triple (food, service, value) is given to the second regressor
- The pairs (food, service) and (food, value) are given to the first regressor
- Among the three scores assigned, the highest one is chosen.

The scores of the regressors are approximated to the nearest integer.

Two versions of Hill Climbing were trained on two different datasets:

- The first one is trained on 80% of the 288 possible cases, leaving 20% as test set.
- The second one is trained on an altered version of the training set above: for each sample a random number of new samples are generated (in our case between 100 and 500), and for each of them the label assigned is altered with a probability that is higher if the average of the sample values is close to one of the two thresholds (as indicated in the previous exercise), in order to reproduce a possible uncertainty of the critic in assigning the vote to a restaurant.

Although the former is more accurate, i.e. it predicts values that correspond exactly to the ground truth, the latter is trained on a dataset that more closely reflects a real situation in which the values assigned to the various restaurants are not always consistent with the guideline provided by the ground truth, with a percentage of average deviation of the assigned values in 7% of cases out of 200 randomly generated datasets. Although the accuracy depends on the dataset and how it is perturbed (in many cases, in fact, the percentage of devation is 0%), in all cases this classifier is always truthful, that is, samples with a '*' in a feature always receive a lower or equal score than the samples that instead have that feature. Experimentally, it was noted that to a sample with a '*' is not assigned a score greater than 2.

The algorithm based on Incentive Compatible Logistic Regression uses two regressors: the first divides the samples with 1 star from those with 2 and 3 stars, the second the ones with 2 stars from those with 3 stars.

The base dataset used for training is generated identically to the second Hill Climbing dataset. However, since the two regressors must be trained with samples having the same number of features, we have chosen to assign the value -1 to the fields in which '*' appears, to differentiate them from samples that have a score of 0 or more on the same feature.

The base dataset is used to generate two different datasets:

- The former assigns label = 0 to samples with 1 star, label = 1 to samples with 2 or 3 stars
- The latter assigns label = 0 to 2-star samples, label = 1 to 3-star samples, thus excluding all 1-star samples.

The two classifiers are then trained respectively with the first and second datasets just described. The classification of a new sample occurs as follows:

- The sample is given as input to the first classifier. If label = 0, it is assigned a 1 star rating.

- If label = 1, the sample is given as input to the second classifier. If label = 0, it is assigned a 2-star rating, otherwise 3 stars are assigned.

The scores assigned to samples deviate from ground truth values in 40% of cases out of 200 tests performed.

In all cases, this classifier is always truthful, that is, samples with a '*' in a feature always receive a lower or equal score than the samples that instead have that feature.

Comparing the two proposed solutions, they are both truthful and have a comparable time when they classify a new sample. Incentive Compatible Logistic Regression, however, has a longer training time and a greater deviation of the predicted values from those of the ground truth, so we are inclined to choose Hill Climbing as the best solution.

Exercise 1 – Final Project

To provide a polynomial implementation of the algorithms for calculating the Shapley Value, a study of the provided papers was carried out. The solution reported within the project was that provided by the paper "Efficient Computation of the Shapley Value for Game-Theoretic Network Centrality", which provides an implementation with computational complexity O (n+m) for the Shapley Degree and Shapley Threshold and O (n*m+(n^2)*log(n)) for the Shapley Closeness.
Regarding the Shapley Closeness, it was necessary to make an implementation choice. The formula requested differed from that reported in the paper as the latter provides an increase of 1 to the dominator which brings f(0) to be equal to 1. In the formula requested, this f(0) is infinite. Consequently it was decided to consider f(0)=f(1)=1 in our case, to respect the decrease constraint of the function f.

In this paragraph, it is shown that the Friedkin-Johnsen dynamic experimentally converges to a stable state. To demonstrate this, dynamics were applied 5 times to all networks provided in the track, then from net_1 to net_17. To differentiate the various tests, the values of belief and stubbornness were randomly chosen for each node as numbers between 0 and 1. In all cases, the dynamics come to a point where the number of stable nodes is equal to the number of nodes in the network, which is just what we want.

Below are the tests on the different networks with the number of cycles required to achieve stability:

	Prova 1	**Prova 2**	**Prova 3**	**Prova 4**	**Prova 5**
Net_1	14	14	14	14	12
Net_2	10	9	11	12	9
Net_3	12	13	14	12	13
Net_4	10	11	10	10	11
Net_5	15	16	15	16	15
Net_6	13	14	13	12	13
Net_7	13	13	12	13	11
Net_8	13	12	13	14	13
Net_9	13	14	14	15	14
Net_10	16	18	16	16	15
Net_11	13	12	11	12	12
Net_12	13	12	13	13	13
Net_13	14	13	14	13	13
Net_14	26	19	23	35	24
Net_15	11	12	12	12	12
Net_16	16	15	15	14	14
Net_17	18	19	23	19	18

Having established that the dynamics converge to a stable state, we wanted to analyze the dynamics on the network net_13, with different values of the stubbornness of the nodes (therefore not randomly generated). In the following table, we report the results of these tests.

Stubborness	Cicli necessari alla convergenza	Tempo di convergenza (s)
0.0	146	360.4
0.1	34	75.9
0.2	22	50.6
0.3	15	34.2
0.4	12	26.2
0.5	10	21.0
0.6	8	16.5
0.7	7	14.2
0.8	6	11.6
0.9	5	10.0
1.0	2	2.5

Also in this case we notice that the dynamics converge for different values of the stubbornness, and it is possible to notice how the number of cycles required for convergence and the convergence time decrease as the value of the stubbornness of the nodes increases.

Below are also the graphs:

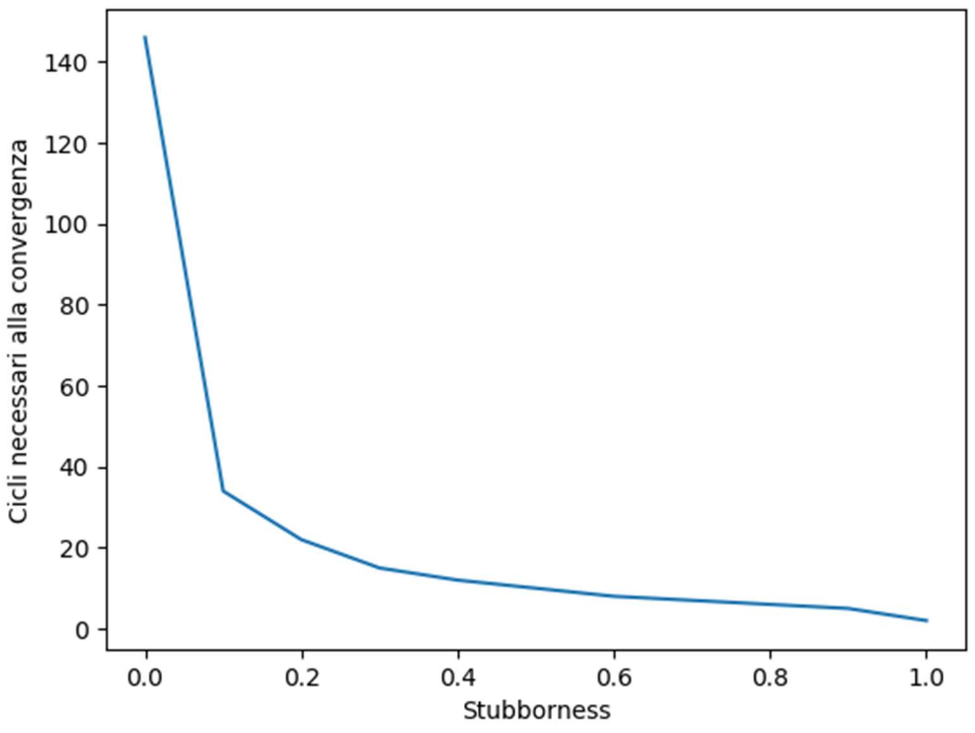


Figura 1: Grafico stubbornness - cicli convergenza

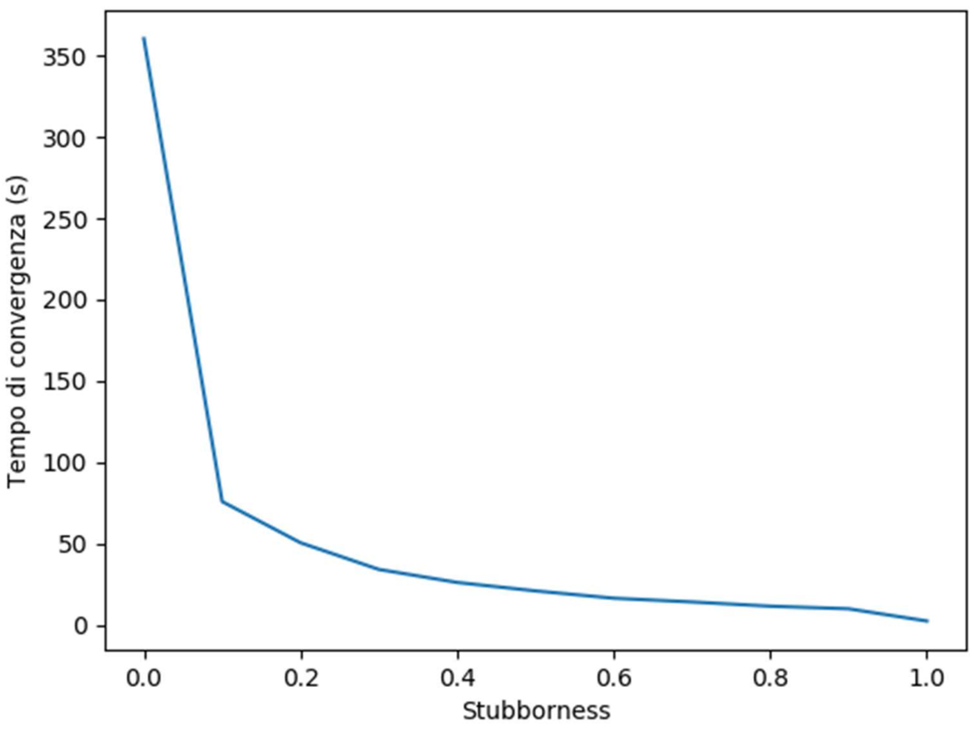
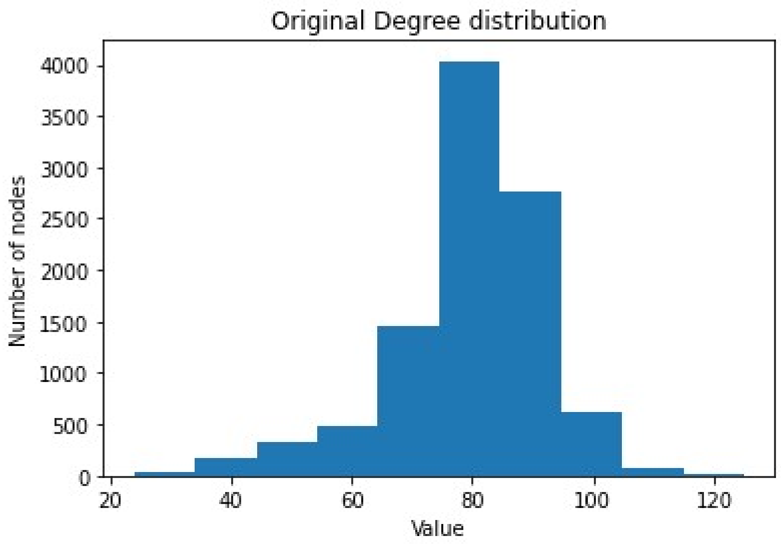


Figura 2: Grafico stubbornness - tempo di convergenza

Exercise 2 – Final Project

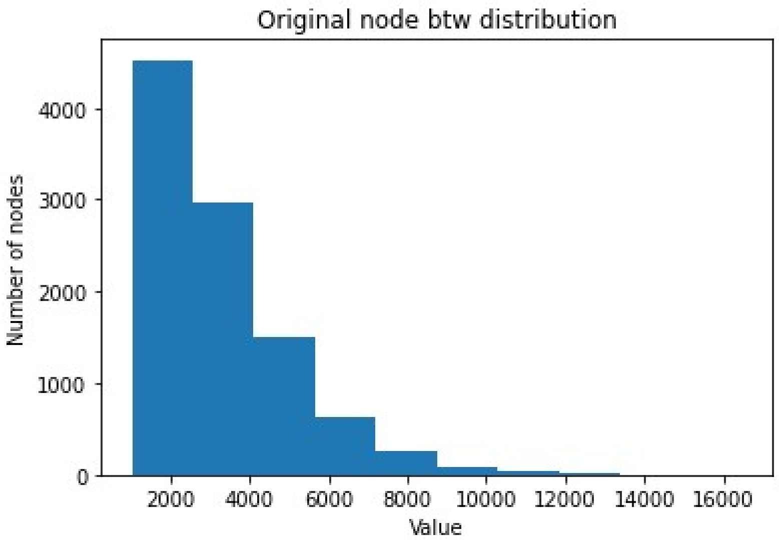
Group 13 has been assigned the net_13 network, which consists of 10000 nodes and about 400,000 edges (397985). To try to understand which model had been used to generate such a network, first the structure of net_13 was analyzed.

First of all, we went to look at the distribution of the degrees of the nodes, represented graphically below:



From the previous graph, it is evident that the distribution of nodes does not follow a power law. This has made it possible to discard (among the models studied in lessons) those models that generate a distribution of degrees that follows a power law. Specifically, the network assumes that the network was not generated by Configuration model, Preferential attachment model, and Affiliation Network model.

Another property that has been analyzed is the diameter of the network. It has been noticed that the network has an estimated diameter of about 4, which would suggest a rather compact network (one could think of a small world phenomenon). This hypothesis can be strengthened by counting the number of triangles of the network, equal to 5756862, a rather large number considering the presence of only 10000 nodes. Another element in favor is the distribution of betweenness within the network, graphically represented below:



It is possible to notice that most of the nodes have a rather low betweenness value, which is around 2000, while a small percentage of nodes have a very high betweenness, which is around 13000.

Among the networks seen in class, the Random Graph and the Generalized Watts-Strogatz model remain to be analyzed. We can remember that for a Random Graph the expected value of the diameter is around ln(n), where n is the number of nodes. In our case, having 10000 knots, we would expect a diameter of about 9. We have previously seen that instead the diameter is much smaller, and stands at an estimated value of 4. This makes us think that the network was not generated as a Random Graph but, also according to the hypothesis of a Small World phenomenon, it is assumed that the network was generated with a Generalized Watts-Strogatz model.

We know that the Generalized Watts-Strogatz requires 3 parameters to be defined, in particular:

- r is the radius of each node (a node u is connected with each other node at distance at most r) - strong ties
- k is the number of random edges for each node u - weak ties
- q, the exponent in the probability formula for inserting a weak tie

Numerous networks were generated with different combinations of these parameters, to try to understand which combination had given rise to the network assigned to us.

The parameters set are shown in the following table:

	r	**k**	**q**
Test 1	3	100	2
Test 2	3	70	2
Test 3	4	70	3
Test 4	4	40	2
Test 5	4	40	1
Test 6	4	40	1.5
Test 7	4	30	1.5
Test 8	4	20	1.5
Test 9	4	25	1.5
Test 10	5	25	1.5
Test 11	5	20	1.5
Test 12	5	10	1.5
Test 13	5	10	2
Test 14	5	9	2
Test 15	5	8	2
Test 16	5	7	2
Test 17	5	6	2
Test 18	5	5	2
Test 19	5	4	2

The parameters that generated a network more similar to the assigned network were as follows:

r = 5
k = 7
q = 2

Exercise 3 – Final Project

The election manipulation algorithm is mainly divided into two files, manipulator.py and selection.py.
In manipulation.py there is the manipulation(G, p, c, B, b) function as required, together with the plurality_voting_rule(p, b) function, which counts the votes for each candidate according to the plurality voting rule, and to the FriedkinJohnsen(G, stubborness, belief) function which performs the FJ dynamics on graph G according to the stubborness and belief passed in input.
In selection.py there is the selector(G, B) function which returns the best B seeds of graph G based on the linear combination of the Shapley values based on the Shapley Degree, Shapley Threshold (imported from the shapley.py file in the exercise1_final folder) and Shapley Closeness. The Shapley Closeness algorithm has been included in the selection.py file because a parallel version has been implemented to minimize execution times.
Therefore, in order to execute the manipulation function, it is necessary to be able to import the functions present in selection.py and exercise1_final/shapley.py.

The various experiments were carried out on different networks (all the networks provided for exercise 2 of the final project and other networks recovered from the internet) and it was possible to experimentally determine how the game-theoretic centrality measures allow obtaining better results than the classic ones. Despite this, each of the three types of Shapley value obtained low results on different types of networks, such as the Shapley Closeness on the net_3 network. On the contrary, their linear combination did not show this type of problem, performing well on every network, in some cases improving them and in others getting very close to the results of the best Shapley Value for that network.
Finally, different k values were tested in the Shapley Threshold, concluding that too high values caused a worsening of the results and low values tended to obtain the same results as the Shapley Degree and did not bring improvements, therefore choosing to use a k equal to 5.

In addition, clustering algorithms were also tested in the various experiments to select the seeds in each cluster in proportion to their size. However, this choice did not lead to improvements, therefore deciding not to include it in the final solution to reduce execution times.