Visualization and analysis of complex networks

Programming Project

Janvier 2018

INF 421 – Conception et analyse d'algorithmes

Programming Project – Visualization and analysis of complex networks

Fan Jiang & Romain Cosson

Introduction

Large graphs can be hard to interpret efficiently. The aim of the project is to study two different tools which are essential to graph analysis: the drawing of a clear layout and the detection of well-connected communities inside the graph we call. In this report we come back linearly on the project and will write in `this font` the elements you will also find in the code. We will present our results on different graphs but will give a special attention to *dtw_592* network in order to allow comparisons between the various implementations that were suggested.

1. Network visualisation

a. The FR91 algorithm

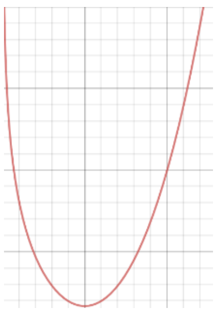
The *spring electrical model* consists in considering the nodes of the graph as charged particles exerting forces one on the other and simulating their movement in this context. Every pair of nodes are repulsed inversely proportionally to their distance (to prevent nodes from superposing themselves) and adjacent nodes are attracted proportionally to the square of their distance (to prevent adjacent nodes from being too distant in the layout).

If we look at things physically, in one dimension, the potential energy of the system formed by two isolated and adjacent vertices only depends on their distance. It goes through a unique minimum in d_{min} which is a stable equilibrium.

$$F_a(d) = \frac{d^2}{K}$$

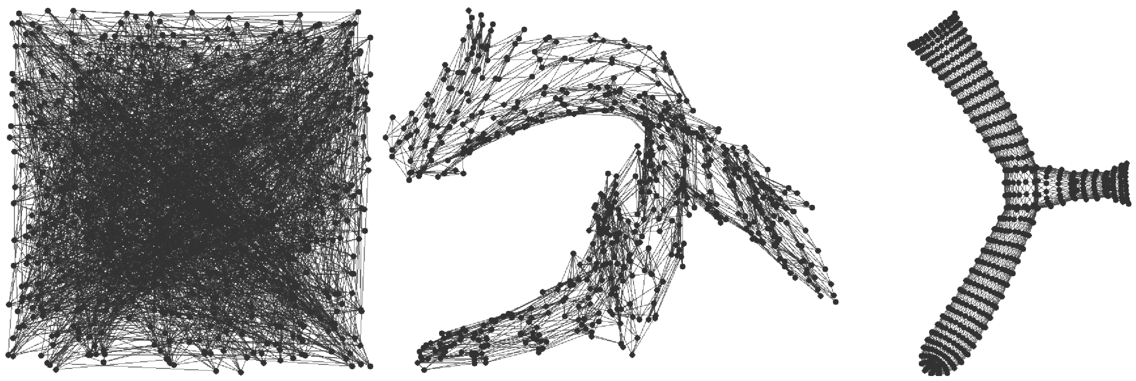
$$F_r(d) = -\frac{CK^2}{d}$$

$$Ep(d) = -CK^2 \ln d + \frac{d^3}{3K}$$



Representation of the potential energy

At each iteration of the FR91 algorithm, this physical model is simulated for a short period of time which decreases with a `temperature` variable. So, according to the forces on each node, the nodes displaced by certain amount of distance in certain directions.



The graph dwt_592 after 0, 70 and 500 iterations of FR91

b. Improving the runtime with Octree

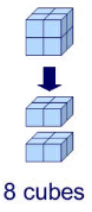
The complexity of FR91 is $O(n^2)$ because it computes at least one force for each pair of vertex.

The idea with FastFR91 is to approximate distant groups of nodes by their barycenter. At each iteration, when computing the repulsive forces on a certain node, we will segment the space in order to consider only $O(\log n)$ barycenters in average. The average complexity of FastFR91 therefore goes down to $O(n \log n)$.

i. The Octree class

Each iteration of the FastFR91 algorithm needs a decomposition of the space in an Octree that is a recursive partition of our space. To compute an Octree we proceeded as follows.

- First we compute a bounding cube (represented by two extremity points) in which every vertices of the graph are contained.
- The construction of Octree which partitions our bounding cube is recursive and as follows:
  - The current Octree represents the bounding cube which partitioned into 8 smaller cubes: its children.
    - If one of the children does not contain any node, it remains `null` and is not partitioned anymore.
    - Else the child is itself partitioned recursively.
  - At each step we calculate recursively the number of vertices contained in the Octree and their barycenter. They will help us compute the value of repulsive forces.



If the repartition of the nodes is balanced, the creation of an Octree with n vertexes should be in $O(n \log n)$. Indeed the runtime of the creation T_n should then verify approximately $T_n = 8T_{\frac{n}{8}} + O(n)$.

ii. The computation of repulsive forces

Let us now focus on the way to efficiently compute a good approximation of repulsive forces which are exerted on a given node `u` by the vertices of an Octree. We do this recursively:

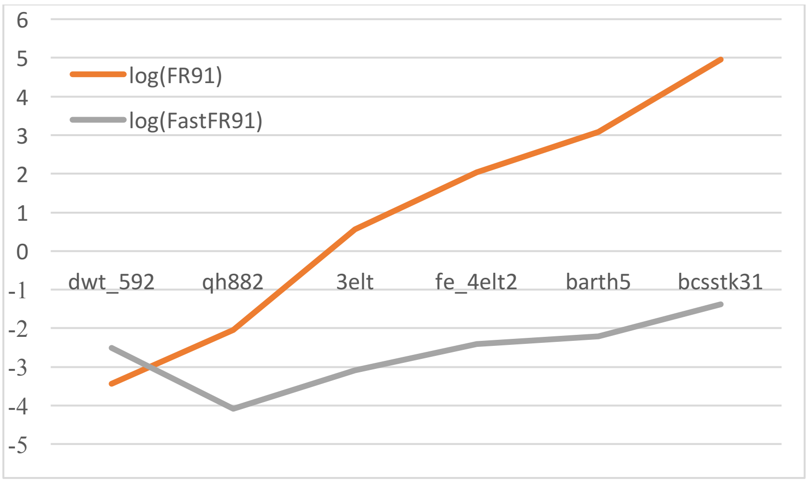
- If the Octree is a leaf (an isolated node) we use the classic computation of repulsive forces between two points.
- If the Octree we are is sufficiently far from `u` (More precisely, that is when the width of the Octree is smaller than a certain constant `theta` times `norm` the distance between `u` and the barycenter of the Octree). We compute the repulsive force assuming that all of the edges contained in the octree are positioned at its barycenter. This means that the more distant the current Octree is from the point `u`, the quickest the algorithm will stops its tree descent for precision.
- If none of the two previous condition is verified, we sum recursively the forces applied by each child of the Octree to our vector `u`.

iii. Results

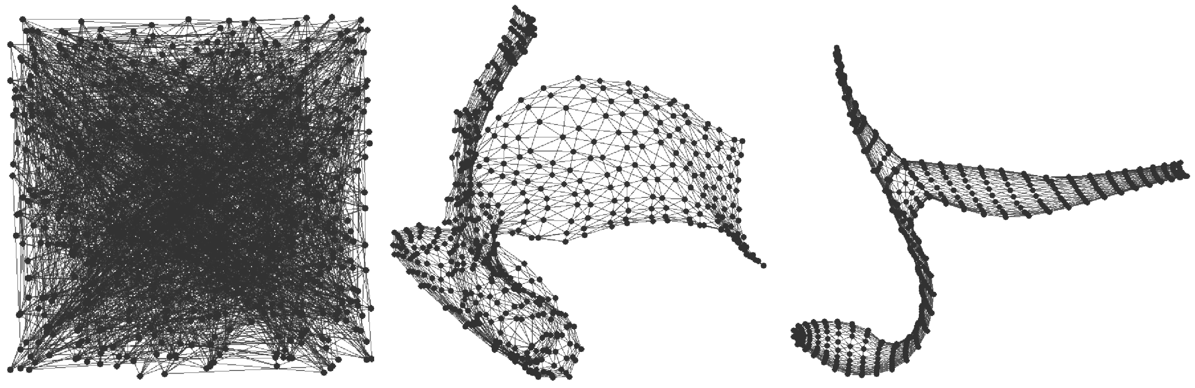
The FastFR91 algorithm has a better complexity and therefore allows us to visualize much bigger graphs. We present here the execution time and its logarithm on graphs of very different sizes.

	Nodes	FR91	ln(FR91)	FastFR91	ln(FastFR91)
dwt_592	592	0,032	-3,43	0,082	-2,50
qh882	882	0,13	-2,02	0,017	-4,07
3elt	4720	1,74	0,55	0,046	-3,07
fe_4elt2	11143	7,7	2,04	0,09	-2,40
barth5	15606	21,7	3,07	0,11	-2,20
bcsstk31	35588	141	4,94	0,25	-1,38

When we plot the logarithm of the execution time for both FR91 (in orange) and FastFR91 (in grey) we observe that the slope of the grey curve is inferior to the slope of the orange curve. This confirms that the asymptotic complexity is improved by FastFR91.



We noted that the limit layout given by FastFR91 is different from the one given by FR91. If we compute the algorithm on the same example *dtw_592*, we obtain the following.



The graph dwt_592 after 0, 70 and 500 iterations of Fast FR91

This is certainly due to the approximations made by the Octree method. We also noted an unexpected behaviour of the graph when we run FastFR91: its barycenter sometimes moves inside the layout. This comportment is not physically realistic because the system of all the points of the graph is subject to no outer forces (Newton's second law). It is due to the fact that with Octree, forces are not exactly symmetric (violation of Newton's third law).

2. Community detection

a. Modularity

Modularity evaluates the quality of a partition of a graph into communities. In order to compute it we stored in a HashMap k_c the total degree of the nodes contained in a community c, and L_c the number of inner edges in c. The following formula therefore allows to compute the modularity of a graph in $O(n+m)$.

$$M(graph) = \sum_{c\,=\,community} \left(\frac{L_c}{L} - \left(\frac{k_c}{2L}\right)^2\right)$$

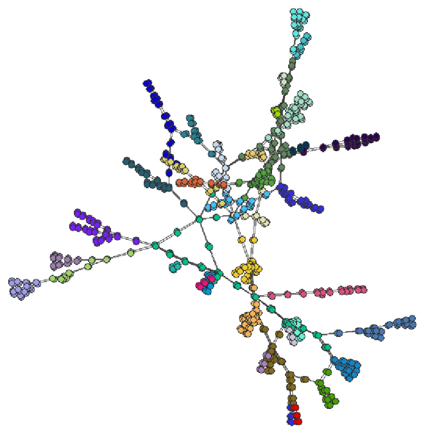
b. Greedy algorithm

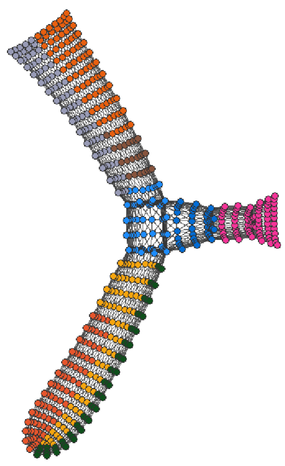
Newman gives in his article an equivalent expression of modularity. It however implies to see an undirected graph as a directed network by doubling each edge into two directed edges in both sides and to interpret the weight between to vertices as the number of edges linking one to the other. Then let e_{ij} be the fraction of edges in the network that connect vertices from group i to those in group j and let $a_i = \sum_j e_{ij} = \frac{k_i}{2L}$. Then $e_{ii} = \frac{L_i}{L}$ and $a_i = \frac{k_i}{2L}$.

$$M = \sum_i (e_{ii} - {a_i}^2)$$

If we merge two clusters i and j into one cluster c, we have $e_{cc} = e_{ii} + e_{jj} + 2e_{ij}$ and $a_c = a_i + a_j$ the gain of modularity is therefore $\Delta M = 2(e_{ij} - a_i a_j)$.

The greedy algorithm consists in modifying step by step our communities starting with isolated nodes and merging at each step the two clusters offering the most important gain for modularity ΔM. To represent clusters efficiently we used a union-find implementation over the array `communities`. At the end of this first step all the nodes have been merged into one single cluster. As we have stored the value of the modularity of our partition at each step we can go back to the moment when it was maximal. To do this we record the merges that are made throughout the steps in a HashMap `configurations` and we redo the union-find over another array stopping exactly when M is maximal.

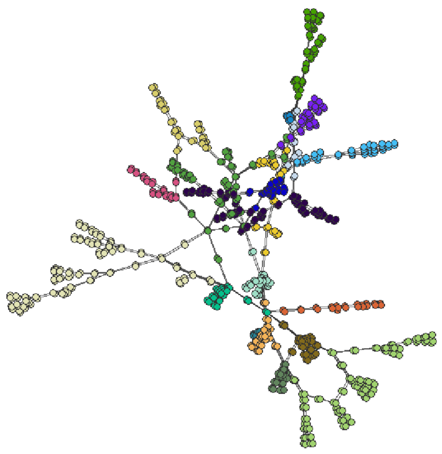

The greedy algorithm on qh882 M = 0.875

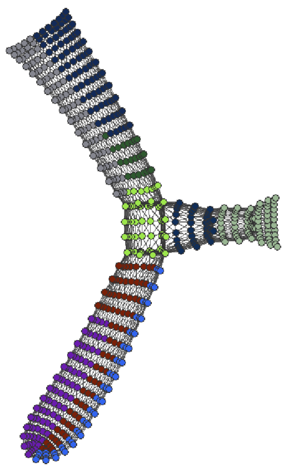

The greedy algorithm on dwt_592 M = 0.74

c. Louvain algorithm

The Louvain algorithm is also based on a step by step modification of a partition and on the expression of ΔM the difference of modularity when we make this modification. If a node i is taken to join the community c of one of its neighbours, the change of modularity is $\Delta M = \frac{k_{i,in}}{2m} - \left(\frac{k_i \Sigma_{tot}}{2m}\right)^2$ where $k_{i,in}$ is the number of edges which link i with nodes of c, k_i the degree of node i and Σ_{tot} is the total degree of the nodes in c. Similarly, we can also calculate the gain of modularity by taking a node i from its community c.

In the first phase of the Louvain algorithm, for each node i, we consider all its neighbors j, and calculate the gain of modularity if we remove i from its community and place it into j's community using the previous formula. We then place the node i in the community which gives the maximum gain of modularity. If no gain of modularity can be achieved, we move on to the second phase. We construct a new graph by merging all nodes that are in a same community. In this way, every node in the new graph is a community in the old graph. Besides, each edge between different nodes in the new graph has a weight, which is equal to the number of edges between two communities in the old graph. Edges between nodes in the same community are considered as self-loops with weight in the new graph. These combined two phases are called one pass. The algorithm will repeat several passes until no further improvement of modularity can be achieved after one pass.


The Louvain algorithm on qh882 M = 0.872


The Louvain algorithm on dwt_592 M = 0.66

d. Results

We can compare the modularity obtained by both methods on different graphs. The Greedy algorithm seems to give a slightly higher modularity than the Louvain algorithm. Note that the difference of modularity does not seem to reduce with the number of nodes in the graph as one could have expected.

	Nodes	Greedy - Modularity	Clusters	Louvain - Modularity	Clusters
ash85	85	0.60	9	0.62	5
dwt_592	592	0,73	9	0,66	7
qh882	882	0,87	39	0,87	20
facebook	1128	0.81	45	0.77	39
3elt	4720	0,86	11	0,83	10
fe_4elt2	11143	0,85	11	0,63	7
barth5	15606	0,89	13	0,78	21
bcsstk31	35588	Out of Memory		0.68	22

Conclusion

This project gave us the opportunity to discover two tools that are absolutely essential in graph analysis. The open-source network analysis and visualization software package Gephi implements this type of methods, and obviously along with many others.