Visualization and analysis of complex networks

Programming Project

Your project must contain a /report/ directory, containing your project report (your report should be **between 4 and 6 pages**, in pdf format);

Janvier 2018

INF 421 – Conception et analyse d'algorithmes

Programming Project – Visualization and analysis of complex networks

Fan Jiang & Romain Cosson

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 (adjacent nodes should not be 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.

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

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

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

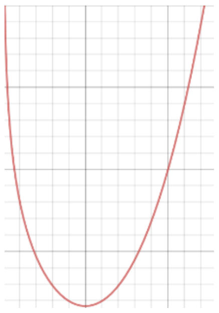


Figure 1: Representation of the potential energy

At each iteration of the FR91 algorithm, the physical model is simulated for a short period of time which decreases with a variable called `temperature`.

- `private Vector_3[] computeRepulsiveForce(Node u)` computes the repulsive forces which are applied to a certain node by going through every other node of the graph.
- `private Vector_3[] computeRepulsiveForce(Node u)` computes the attractive forces applied on a certain node by going through every neighbour of `u`.

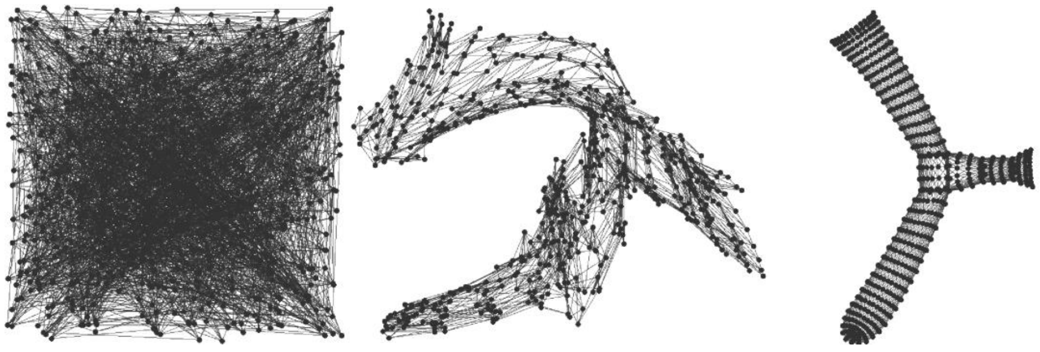


Figure 2: The graph dwt_592 after 0, 70 and 500 iterations of FR91

b. Improving the runtime with octrees

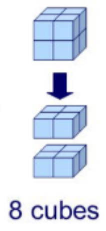
The complexity of FR91 is $O(n^2)$ because it computes at least one force for each pair of vertex.

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

i. The octree class

At each iteration of the FastFR91 algorithm we should compute a decomposition of the space in an Octree.

- First `public static Point_3[] compute3DBoundingBox(ArrayList<Node> vertices)` computes a 3D bounding box (represented by two extremity points) in which every vertices of the graph is contained.
- Then `Octree octree = new Octree(this.g.vertices,0,boundingBox));` constructs the octree which will partition our bounding box. The construction of octree is recursive and as follows:
    - The current octree represents the bounding cube which partitioned into 8 smaller cubes: it's children.
        - If one of the children does not contain any node, it remains `null` and is not partitioned anymore.
        - Else the child is himself partitioned recursively.
    - At each step we calculate `public int num_vertice` and, `Point_3 barycenter` respectively 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 $T_n = 8T_{\frac{n}{8}} + O(n)$.

ii. The computation of repulsive forces

Let us now focus on the implementation of the function `private Vector_3 computeRepulsiveForce(Node u, Octree octree)`. For a given node the function goes down in the octree and stops:

- If it has attained a leaf (an isolated node)
- If it is considering an octree which is sufficiently far from `u`. That is when `octree.width/norm <= this.theta`, `norm` being the distance between `u` and `octree.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.

iii. Results

The FastFR91 algorithm improves complexity and allows us to visualize much bigger sets. However we should note that the limit layout it gives is different from the one given by FR91. If we compute the algorithm on the same example as previously, we obtain the following.

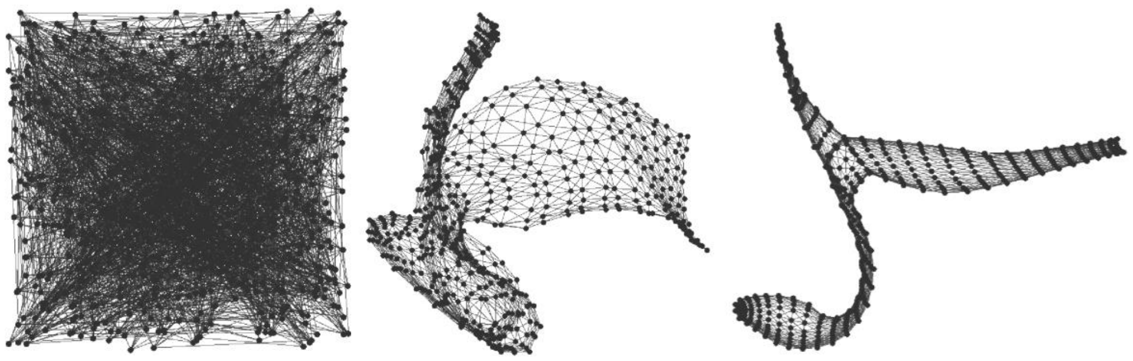


Figure 2: The graph dwt_592 after 0, 70 and 500 iterations of Fast FR91

This is certainly due to the approximations made by the octree method. We wan also note an unexpected behaviour of the graph: it's barycenter sometimes moves inside the layout. This comportment is not physically possible 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 octrees forces are not exactly symmetric (violation of Newton's third law). To counter this problem we could chose repulsive forces that are proportional to $\frac{1}{d^2}$ (like the classic gravitational force). In this case, the principle of Gauss says that the approximation with the barycenter is exact. (I need to check this up).

2. Community detection

a. Modularity

Modularity evaluates the quality of the partition of a graph.

`public double computeModularity(AdjacencyListGraph graph, int[] communities)` computes the modularity of the partition `communities` of `graph`. For each community `k` we store in a `HashMap kc` the total degree of the nodes in the community and in the `HashMap lc` the number of inner edges (which should note be counted twice). We can compute the modularity of the graph in $O(n+m)$.

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

b. Greedy algorithm

Newman in his article gives a slightly different expression of modularity which we have chosen to keep. In the article e_{ij} is the fraction of edges in the network that connect vertices in group i to those in group j.

$$a_i = \sum_j e_{ij} \qquad and \qquad M = \sum_i (e_{ii} - {a_i}^2)$$

Therefore $e_{ii} = \frac{L_c}{L}$ but we do not have $a_i = \frac{k_c}{2L}$ because outer nodes will only count for half as much in the expression $\frac{k_c}{2L}$ as they will in a_i.

If we apply the greedy algorithm to our graph we obtain the following result.

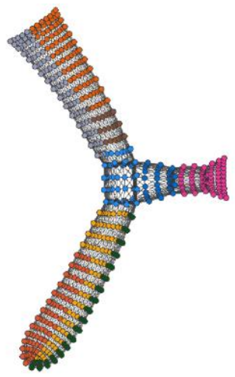


Figure 2: The greedy algorithm on dwt_592 modularity = 0.74

c. Louvain algorithm

The Louvain algorithm has better runtime performances than the greedy algorithm but it seems to give partitions with lower modularity. To compare it, we give a representation of its result on dwt_592.

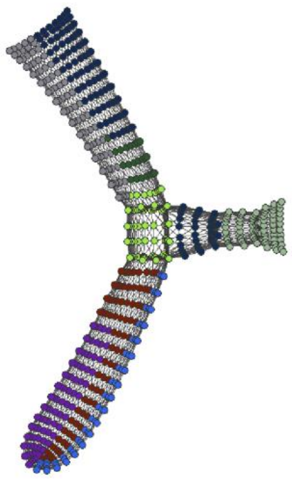


Figure 2: The Louvain algorithm on dwt_592
modularity = 0.66

3.