VISUALIZATION AND ANALYSIS OF COMPLEX NETWORKS

(PROGRAMMING PROJECT, OCTOBER 2017)

LUCA CASTELLI ALEARDI (LIX)

Abstract. Networks are important objects in several applications domains (ranging from computer science to biology and to social sciences), since they provide a mathematical model for representing the interactions (the network links) between the entities (the network node) of a complex system (such as proteins-to-proteins interations, or interactions in a social network). The problem of computing a visual representation of a network is a challenging problem that has attracted a lot of attention in the past 3 decades: having a pleasing layout is a first crucial step for understaning the structure of a network. Another important problem is the detection of communities (locally dense sub-graphs) which is very relevant in network analysis: detecting a dense cluster of nodes help to understand how human communities interact in social networks, or still to understand some biological functions encoded in cellular networks.

The goal of this project is to provide fast tools for dealing with the visualization and community detection problems for large complex networks.

Requirements: this project requires a basic knowledge of discrete maths and elementary geometry

Key words: graph theory, computational geometry.

Author: Luca Castelli Aleardi (`amturing@lix.polytechnique.fr`)

Url: website (with related material)
`http://www.lix.polytechnique.fr/~amturing/Teaching/Project_NetworkVisualization/Project.html`

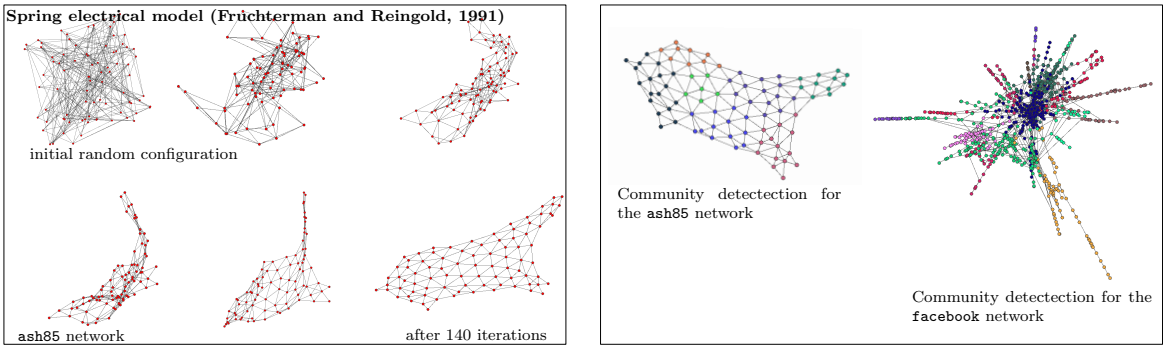

Figure 1. (Left) Force-directed layout of the `ash85` graph. (Right) Coomunity detection via modularity optimization.

1. Network visualization

As mentioned in the abstract, our goal is to take as input a *graph* (also called *network*) and to produce a 3D layout where the *edges* (also called *links*) connecting two vertices (also called *nodes*) are drawn as straight-line segments. In particular, we want to make use of the *force-directed* paradigm: the graph is assumed to represent a physical system, with forces interacting between its nodes (Fig. 1 illustrates the layout obtained after 140 iterations of the `FR91` algorithm).

In this work you are asked to implement a few solutions for the fast computation of nice network layouts: a comprehensive presentation of this topic can be found in [3].

1.1. Force-directed layouts: the spring electrical model.

We are given as input an arbitrary graph G with n vertices and we want to compute a *layout* (a drawing) of G in 3D using the *spring-electrical model* introduced by Fruchterman and Reingold in [2]. As in [2] we compute attractive forces (between adjacent vertices) and repulsive forces (for any pair of vertices) acting on vertex

```
{ initialisation }
function f_r(x, w) := begin return −Cwk²/x end
function f_a(x) := begin return x²/k end
t := t_0;
Posn := NewPosn;
iterations := 0

for iterations < m begin

    for v ∈ V begin
        OldPosn[v] = NewPosn[v]
    end

    for v ∈ V begin
        { initialise Θ, the vector of displacements of v }
        Θ := 0;

        { calculate (global) repulsive forces }
        for u ∈ V, u ≠ v begin
            Δ := Posn[u] − Posn[v];
            Θ := Θ + (Δ/||Δ||) · f_r(||Δ||, |u|);
        end

        { calculate (local) attractive/spring forces }
        for u ∈ Γ_v begin
            Δ := Posn[u] − Posn[v];
            Θ := Θ + (Δ/||Δ||) · f_a(||Δ||);
        end

        { reposition v }
        NewPosn[v] = NewPosn[v] + (Θ/||Θ||) · min(t, ||Θ||);


    end
    iterations + +
    { reduce the temperature to reduce the maximum movement }
    t := cool(t);
end
```

Spring electrical model (by Fruchterman and Reingold, 1991)

$$F_a(u) = \sum_{(u,v)\in E} \frac{\|\mathbf{x}(u)-\mathbf{x}(v)\|}{K}(\mathbf{x}(v)-\mathbf{x}(v))$$

$$F_r(u) = \sum_{v\in V, v\neq u} \frac{-CK^2(\mathbf{x}(v)-\mathbf{x}(u))}{\|\mathbf{x}(u)-\mathbf{x}(v)\|^2}$$

`ash85` network

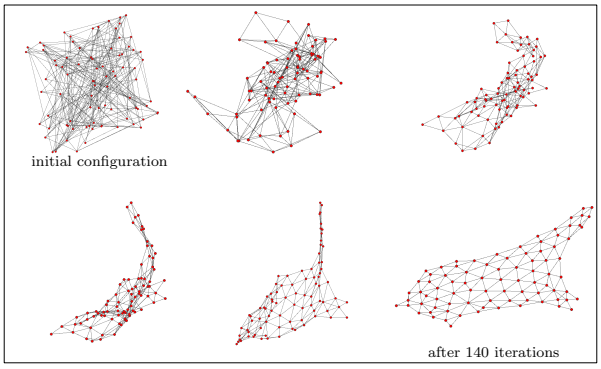

FIGURE 2. *Spring electrical layout*: pseudo-code and example illustrating the `FR91` algorithm.

u, defined by:

$$F_a(u) = \sum_{(u,v)\in E} \frac{\|\mathbf{x}(u)-\mathbf{x}(v)\|}{K}(\mathbf{x}(v)-\mathbf{x}(v)),\ F_r(u) = \sum_{v\in V, v\neq u} \frac{-CK^2(\mathbf{x}(v)-\mathbf{x}(u))}{\|\mathbf{x}(u)-\mathbf{x}(v)\|^2}$$

where the values C (the strength of the forces) and K (the optimal distance) are scale parameters (given as input constants).

The algorithm to implement proceeds in an iterative manner, computing at each step the displacement of each vertex u, that depends on the forces exerced on u (the pseudo-code is illustrated in Fig. 2). The algorithm terminates after a given number of iterations: the resulting layout corresponds to an approximation of a local minimum of the sping energy.

The class `FR91Layout` provides the main methods allowing to initialize and implement the `FR91` algorithm.

Question 1 [4 points]. From the implementation point of view, you are asked to complete the class `FR91Layout` that

- receives as input a graph G,
- computes a layout running the `FR91` algorithm, with exact force computation: repulsive forces are computed for any pair of vertices in the graph.

The Java code provided with this project allows you to open and produce a 3D layout of the input graph stored in MTX format (class `NetworkLayout`), as illustrated in Fig. 2. The classes provided for this project make use of *adjacency lists* to represent the graph: in particular, they provide methods for reading and building an adjacency graph data structure from an input text file in MTX format (see the Appendix for more details).

1.2. **Fast approximation of repulsive forces, via octrees.** The main drawback of the algorithm of previous section is that the exact computation of repulsive forces requires a huge amount of computational resources (for computing the $O(n^2)$ repulsive forces).

In order to decrease the the time complexity, one can approximate repulsive forces, making use of a recursive space decomposition base don *octrees*. The main idea is that, at each iteration, the vertex locations are first partitioned using an *octree*: this is a recursive space decomposition that allows to perform efficient point locations (see Fig. 3-left). When calculating the repulsive forces acting on a vertex u, the vertices which are far from u (above a given distance) are approximated by their barycenter. The octree data structure allow to retrieve the vertices lying in a given range in

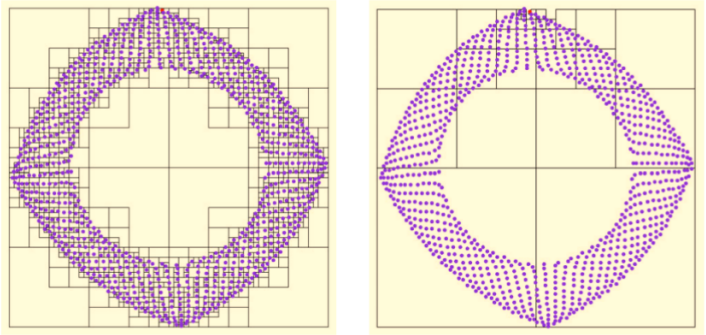

FIGURE 3. Fast approximate computation of repulsive forces via octrees (image by Yifan Hu [3]).

a very fast way, leading decrease the complexity for the force calculation from $O(n^2)$ to $O(n \log n)$ in practice. For more details, we refer to [3] (pages 10-12, Section 4).

Question 2 [6 points]. From the implementation point of view, you are asked to complete the class `FastFR91Layout` that

- given input a graph G, computes a layout using the `Fast FR91` algorithm, with approximate force computation: repulsive forces between nodes at large distances are approximated using octrees space decompositions.

2. COMMUNITY DETECTION

The problem of *community detection* consists in partitioning the graph nodes into clusters which are locally dense sub-graphs. Intuitively, we want to identify clusters (communities) such a given node has high probability of being connected to other nodes in its cluster, while having a small number of links towards different clusters.

2.1. **Modularity.**

In order to evaluate the quality of a partition of a graph, we have first to define a quantitative measure. Among the numerous existing metrics, we adopt the *modularity of a community* which measures the density of edges insides a community as compared to the number of edges linking different communities. To be more precise, for a community C_c having L_c inner edges we define its *modularity* as:

$$M_c = \frac{L_c}{L} - \left(\frac{k_c}{2L}\right)^2$$

where L is the number of edges of the input graph G, and k_c is the total degree of nodes in the community c.

We can define the notion of modularity for a graph partitioned into n_c communities, summing the modularities over $\{C_1, C_2, \ldots, C_{n_c}\}$, as done below:

$$M(G) = \sum_{i=1}^{n_c} M_i$$

The notion of modularity is extremely relevant for the problem of community detection: in particular, it is easy to see that higher values of the modularity correspond to a better community partition (see Fig. 4 for an illustration on a small example). For instance, if the whole graph belongs to the same community, then the modularity is $M = 0$; or if any node belongs to a different community ($n_c = n$) then $M < 0$.

Question 3 [2 points]. From the implementation point of view, you are asked to complete the class `CommunityDetection` that

- given an input a graph G, and a partition of G into n_c communities
- computes the modularity of the graph (complete the method `computeModularity()`)

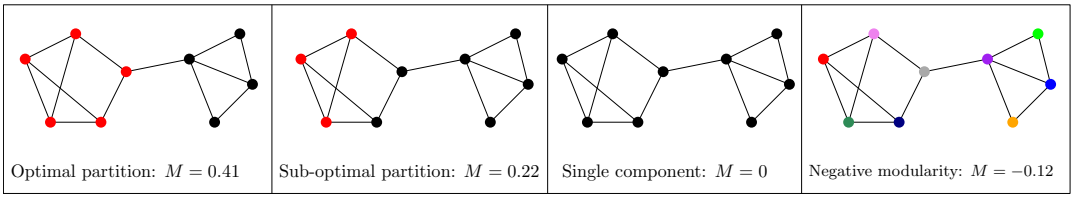

FIGURE 4. Modularity for several partitions of a graph with 9 nodes (example by A. Barabasi).

2.2. **Community detection: the greedy algorithm.** Several algorithms aim to solve an optimization problem using the modularity as objective function.

The *greedy* algorithm [4] proceeds in an iterative way by joining pairs of communities is this operation increases the modularity of the partition. More precisely, the algorith starts with $n_c = n$ communities (each vertex defines a community). For each pair of connected communities C_i and C_j, we compute the possible gain ΔM_{ij} one could get by merging the two communities C_i and C_j. The algorithm perform the merge of the two communities corresponding to the largest gain. This last step is repeated until all nodes are belonging to a single community: at each such step one has to store the value of the modularity M for the corresponding partition.

At the end the algorithm returns the partition for which the value of M is maximal (we refer to [4] for a more detailed presentation).

Question 4 [5 points]. From the implementation point of view, you are asked to complete the class `GreedyClustering` containing

- a method `int[] computeClusters(AdjacencyListGraph g)` that computes and returns a partition of the input graph into communities.

2.3. **The Louvain algorithm.** The last task of this project consists in implementing the so-called *Louvain* algorithm [1], whose runtime performances outperform the ones of the greedy algorithm, allowing us to compute community partitions even for very large graphs.

The main idea is to optimize an objective function (the modularity) performin local changes: the algorithm proceeds in an iterative way, computing at each run two *steps*.

In the first step, the algorithm starts putting each node in a distinct community (so $n_c = n$ at the beginning). For each node v_i, we compute the change in modularity corresponding to moving the node v_i in the community of one of its neighbors v_j: this value (the change in modularity) can be computed efficiently according to a formula given in [1]. Then node v_i is moved to the community for which the gain in modularity is the largest (v_i remains in the same community is the gain is not positive). This process is repeated until no further improvement can be achieved (all nodes remain in their own community).

In a second step, the communities are aggregated in order to obtain a new (smaller) graph: the nodes in the same community merge into a single node (during this process self-loops are created: they come from edges inking nodes in the same community).

The concatenation of the first and second step is called a *pass*: the algorithm continues processing the graph performing a sequence of passes, until no further increase of the modularity is possible.

We refer to [1] for a detailed explanation of this algorithm.

Question 5 [7 points]. From the implementation point of view, you are asked to complete the class `LouvainClustering` containing

- a method `int[] computeClusters(AdjacencyListGraph g)` that computes and returns a partition of the input graph into communities.

3. EVALUATION CRITERIA

The evaluation of this project will take care of several criteria, including:

- the quality of the oral presentation (the use of slides is strongly suggested);
- the clarity of the provided code (your code should be well organized and commented, and include a clear definition of input and output parameters for all methods, according to Javadoc speficications);
- the code should be correct (it should compile and return the ŕight result for each question) and should be as efficient as possible;

- the quality of the report: the report should include a clear presentation of the results obtained with your programs (a short descrption of experimental results and runtime performances must be included);
- submission guidelines should be respected (see Moodle page for the PI);
- submission deadlines **must** be respected.

4. Appendix

4.1. **Geometric objects in 2D and 3D space.** The `Jcg` library provides a collection of classes for representing geometric objects in 2d and 3d space (points, vectors, ...), together with their corresponding geometric predicates and construction.

4.2. **Reading graphs and visualizing networks layouts.** The java code provided with this project allows you to read the input graph (in MTX format, stored as a text file) using the library `TC` (developed at Ecole Polytechnique, see the website of the course INF311). You can run the class `NetworkVisualization` (with one or two parameters) to get a layout of the network: is geometric coordinates are not provided, the vertex locations are placed at random in an unit cube. This class also provides an user interface for running all layout and community detection algorithms: to test your modularity computation, you can produce a random community partition (press `r` button). To run an iteration of the force-directed methods, just press the `c` button (for the `FR91` layout), or the `f` button for the fast version with octrees.

References

[1] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008, 2008.

[2] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw., Pract. Exper.*, 21(11):1129–1164, 1991.

[3] Yifan Hu. Efficient, high-quality force-directed graph drawing. *The Mathematica Journal*, 10(1), 2006.

[4] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Phys. Rev. E*, 69:066133, 2004.