Automatic detection of community structures in networks

UCL
Université
catholique
de Louvain

Group 7
Julien Odent
Michael Saint-Guillain

November 26, 2012

This is the report associated to the project of INMA1691, in which we focus on the automatic detection of community structures in (large) networks, and more specifically on the Louvain method developed by Blondel *et al* in [3].

1 Introduction

In the study of complex networks, a network is said to have community structure if the nodes of the network can be easily grouped into sets of nodes (or *communities*) such that each set of nodes is densely connected internally. This implies that the network divides naturally into groups of nodes with inner dense connections and sparser connections between groups (source: *Wikipedia*). While in general either *overlapping* and *non-overlapping* sets of communities can be considered, we focus here on the particular case of *non-overlapping* community finding problem.

In this report, we focus on the Louvain method developed by Blondel *et al* in [3] for automatic detection of community structures in (large) networks. Our contribution to this work is both material and theoretical. This project first aims at providing a *Python* module implementation of the algorithm, which is usable out of the box. Secondly, we also provide a general analysis of the Louvain method and discuss some heuristic improvements to the algorithm described by Blondel *et al.*

This report is organized as follows: in section 2 we review some of the traditional methods of hierarchical clustering for community finding. We then introduce in section 3 the notion of *modularity* measure for comparing the quality of a network partitioning as described in [2]. Whereafter in section 4 we can finally describe the method developed in [3] (i.e. the Louvain method) and the heuristic improvements on which we focused, together with experimental results obtained with our module on some reference test sets. Finally, a description of how to use the *PyLouvain* Python module that we provide is given in the appendix.

2 Traditional methods: hierarchical clustering

We first take a look at methods presented as "traditional" in [1], i.e. hierarchical clustering methods; many other methods have been developed in data mining and machine learning fields (e.g. standard K-means).

Methods for hierarchical clustering aim at building a dendrogram, either in a agglomerative or a divisive way. A dendrogram (see example on figure 1) is a tree diagram frequently used to illustrate the arrangement of the clusters produced by hierarchical clustering. In a hierarchical clustering the number k of clusters is not provided a priori; one has to choose the appropriate number, based on the dendrogram obtained. The correct choice of k is often ambiguous, with interpretations depending on the shape and scale of the distribution of points in a data set and the desired clustering resolution of the user. In addition, increasing k without penalty will always reduce the amount of error in the resulting clustering, to the extreme case of zero error if each data point is considered in its own cluster (i.e., when k equals the number of data points, n). Intuitively then, *the optimal choice of* k *will strike a balance between maximum compression of the data using a single cluster, and maximum accuracy by assigning each data point to its own cluster.* If an appropriate value of k is not apparent from prior knowledge of the properties of the data set, it must be chosen somehow. However, finding the correct k is then not trivial,

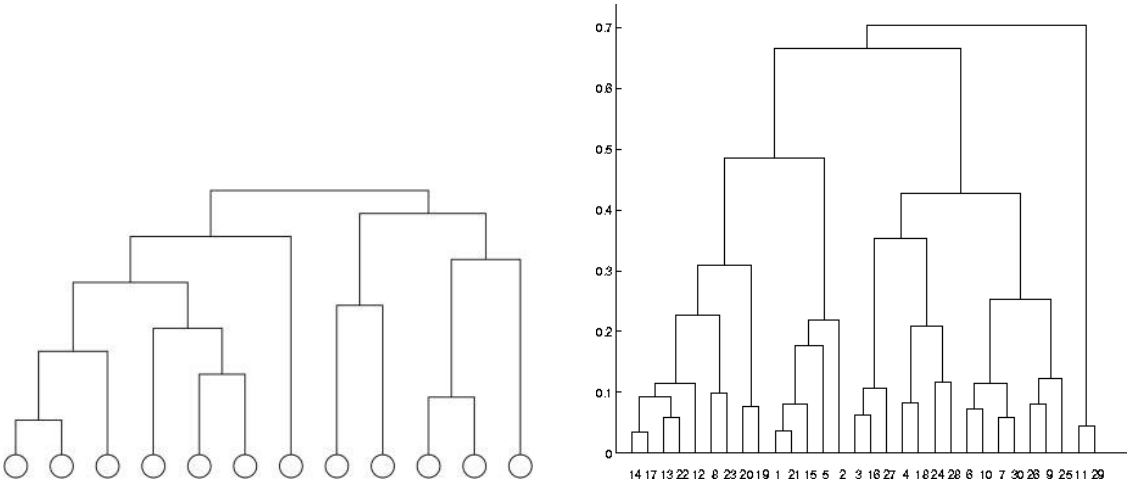

Figure 1: Examples of dendrograms obtained by hierarchical clustering. The circles represent the vertices in the network, and the tree shows the order in which they join together to form communities. In most cases, dendrograms are annotated with the weights of the connections between every clusters (shown on the right figure).

and beyond the scope of this work.

We will now briefly describe the main traditional methods for hierarchical clustering. Since clustering methods always rely on a given notion of distance between the individuals (or nodes), we first need to give a general description (some) of the different distance measures that can be used.

Distances measures in graphs Girvan an Newman describe in [1] several definitions of the weight between two nodes in *unweighted* graphs; one possible definition of the weight is the number of node-independent paths between vertices [1]. Another possible way to define weights between vertices is to count the total number of paths that run between them (all paths, not just those that are node-independent) [2]. In *weighted* (undirected) graphs, the distance between two nodes is simply the total weight of the shortest path between those two nodes.

Hierarchical clustering methods are divided in two categories: *bottom-up* (or *agglomerative*) methods, and *top-down* (or *divisive*) methods.

[1] Two paths that connect the same pair of vertices are said to be node-independent if they share none of the same vertices other than their initial and final vertices. One can similarly also count edge-independent paths.

[2] However, because the number of paths between any two vertices is infinite (unless it is zero), one typically weights paths of length l by a factor α^l with α small, so that the weighted count of the number of paths converges

Agglomerative methods

In an *agglomerative* or *bottom-up* method, the algorithm aims at building the dendrogram by merging clusters two-by-two, starting with one cluster per vertice until there remains only one cluster. At each step, we find the two closest clusters and we merge them. An intuitive example of steps in agglomerative method is given on figure 2.

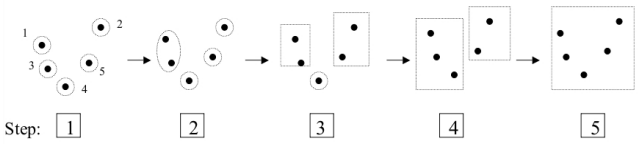

Figure 2: Bottom-up construction

Aggregation criteria In addition to the distance between two nodes of a graph, a hierarchical clustering method also needs a criterion to measure the distance between two *clusters*. There are three principal criteria:

- *Single linkage*: the distance between two clusters is computed as the distance between the two closest elements in the two clusters

$$D\left(C_1, C_2\right) = \min_{x \in C_1,\, y \in C_2} d(x, y)$$

- *Complete linkage*: the distance between two clusters is computed as the distance between the two farthest elements in the two clusters

$$D\left(C_1, C_2\right) = \max_{x \in C_1,\, y \in C_2} d(x, y)$$

- *Average linkage*: the distance between two clusters is computed as the average distance between objects from the first cluster and objects from the second cluster

$$D\left(C_1, C_2\right) = \frac{1}{|C_1|.|C_2|} \sum_{x_1 \in C_1} \sum_{x_2 \in C_2} d(x_1, x_2)$$

Single linkage criterion is sensible to the chaining effect, thus average linkage criterion is often privileged.

In addition to those three common criteria, another very often used criterion is *Ward's criterion*, computing the difference in the *total within-group inertia* (or equivalently *between-group inertia*) when merging two clusters (in a bottom-up fashion) or dividing a cluster (in a top-down fashion). One can prove (see [7]) that, defining the *total inertia* I_{tot} of the entire cloud of points as the sum of squared distances between each point to the center of mass, I_{tot} is constant and equal to $I_W + I_B$; thus after merging two clusters I_B decreases and I_W increases equivalently. **Ward's method** thus merges

two clusters such that $\left|I_W^t - I_W^{t+1}\right|$ is minimum, or equivalently $\left|I_B^t - I_B^{t+1}\right|$ is minimum. This method is more well suited for clouds of points rather than graphs, since one would need to define the notion of center of mass for a set of nodes. However, it yet provides an interesting introduction to methods in which, as in the Louvain method we describe in section 4, one defines its aggregation criteria in terms of the difference, or gain, of a specific action according to a holistic measure of the given system (rather than a local measure, as the three we described above).

Divisive methods

In an *divisive* or *top-down* method, a cluster is partitioned in two parts at each step. Starting with one cluster containing all the vertices, the algorithm processes exactly in the opposite way of the agglomerative method: given a set of cluster, it determines the cluster for which a partitioning into two clusters will provide the best separation between individuals (or vertices) of the two clusters, given a *separation criterion.* [3]

An example of divisive method is the *minimum spanning tree clustering* (see figure 3), whose first computes a minimum spanning tree $G_{MST}(V, E)$ containing all vertices, and then builds the dendrogram by iteratively removing from E the edges with the highest weight; it is easy to see that this method implicitly uses the single linkage aggregation criterion[4].

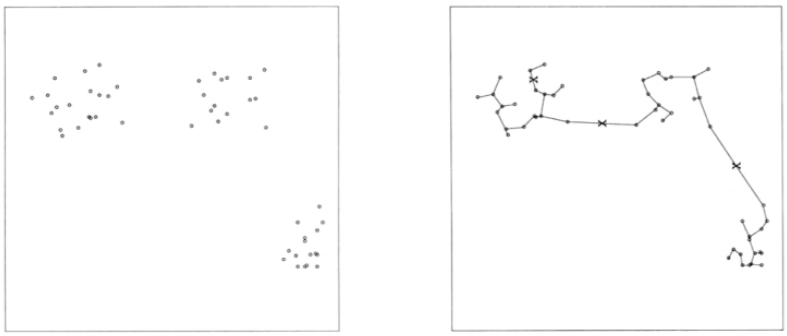

Figure 3: Example of minimum spanning tree

[3] Of course, the criteria described for agglomerative methods (i.e. aggregation criteria) are in general not suitable in a divisive hierarchical clustering fashion since they might require to enumerate each potential cluster in each current cluster in order to compute those inter-cluster distances.

[4] While in general aggregation criteria cannot be used in a divisive hierarchical clustering fashion, the minimum spanning tree clustering uses the *minimum spanning tree underlying structure of the graph* as a heuristic for reducing the space of solutions; the number of potential sub-clusters at each state of the search (i.e. the branching factor in artificial intelligence terms) is then linear in terms of the number of nodes, and aggregation criteria can be computed for each potential cluster.

Edge "Betweenness" and Community Structures

Girvan an Newman (2002) developed in [1] an alternative method for community detection in unweighted graphs, in order to overcome the limitations of hierarchical clustering methods. They define the edge betweenness of an edge as the number of shortest paths between pairs of vertices that run along it. If there is more than one shortest path between a pair of vertices, each path is given an equal weight such that the total weight of all of the paths is unity. If a network contains communities or groups that are only losely connected by a few intergroup edges, then all shortest paths between different communities must go along one of these few edges. Thus, the edges connecting communities will have high edge betweenness. By removing these edges, we separate groups from one another and so reveal the underlying community structure of the graph.

Link with minimum spanning tree clustering It is easy to see that the method suggested by Girvan an Newman (2002) is actually an unweighted graphs variant of the minimum spanning tree clustering (described in section 2), using the edge betweenness centrality as aggregation criterion[5]; indeed, it is equivalent to performing a minimum spanning tree clustering on a graph with edge weights being equals to their corresponding edge "betweenness".

3 Modularity

The problem spotted in [2] is as follows: suppose that we are given some network and that we want to determine whether there exists any natural division of its vertices into non-overlapping groups or communities, where these communities may be of any size. Perhaps the most obvious way to tackle this problem is to look for divisions of the vertices into two groups so as to minimize the number of edges running between the groups.

Although this approach is the one most often adopted in the graph-partitioning literature, the problem is that the sizes of the communities are not normally known in advance. If community sizes are unconstrained then we are, for instance, at liberty to select the trivial division of the network that puts all of the vertices in one of our two groups and none in the other, which guarantees we will have zero intergroup edges.

Newman (2006) defines in [2] the measure of *modularity* as, up to a multiplicative constant, *the number of edges falling within groups minus the expected number in an equivalent network with edges placed at random.* The modularity can be either positive or negative, with positive values indicating the possible presence of community structure. Thus, one can search for community structure precisely by looking for the divisions of a network that have positive, and preferably large, values of the modularity. Blondel *et al* in [3] define the modularity of a partition as a scalar value between -1 and 1 that measures the density of links inside communities as compared to links between communities.

[5] In the case of a weighted graphs, it might be useful to also take into account the betweenness centrality of edges; one can then wonder about the relevance of considering the edge's weight times its betweenness measure as aggregation criterion on weighted graphs.

Suppose our network contains n vertices. For a particular division of the network k communities, let $c_i = l$ if vertex i belongs to community l . Let A be the adjacency matrix such as A_{ij} represents the weight of the edge between i and j , and $k_i = \sum_j A_{ij}$ is the sum of the weights of the edges attached to vertex i . The expected number of edges between vertices i and j if edges are placed at random is then $k_i k_j / 2m$, where $m = \frac{1}{2}\sum_i k_i$ is the total number of edges in the network. Thus the modularity Q is given by the sum of $A_{ij} - k_i k_j / 2m$ over all pairs of vertices $i, , j$ that fall in the same group. One can then express the modularity to be maximized as:

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(c_i,\, c_j) \tag{1}$$

, where the δ-function $\delta(u, v)$ is 1 if $u = v$ and 0 otherwise. The Q-value of a graph partitioning can then be used as a measure of the quality of the partitioning, while for instance comparing several different community detection algorithms.

The Method of Optimal Modularity

The method developed in [2] tries to find the c_i's values which maximize this modularity. Although a large number of alternative heuristic methods have been investigated, such as simulated annealing [4], greedy algorithms [5] and extremal optimization [6], Newman uses an approach based on the eigenvectors of a characteristic matrix for the network, which called the *modularity matrix* $B_{ij} = A_{ij} - \frac{k_i k_j}{2m}$. The study of this approach is however out of the scope of this paper.

4 The Louvain method

We now describe the Louvain method developed by Blondel *et al* [3] which finds high modularity partitions of large networks in short time and that unfolds a complete hierarchical community structure for the network, thereby giving access to different resolutions of community detection (i.e. building a dendrogram). Contrary to all the other community detection algorithms, the network size limits that we are facing with our algorithm are due to limited storage capacity rather than limited computation time.

Description Authors in [3] describe their algorithm as follows: the algorithm is divided in two phases that are repeated iteratively. Assume that we start with a weighted network of N nodes. First, we assign a different community to each node of the network. So, in this initial partition there are as many communities as there are nodes. Then, for each node i we consider the neighbors j of i and we evaluate the gain of modularity that would take place by removing i from its community and by placing it in the community of j. The node i is then placed in the community for which this gain is maximum (in case of a tie we use a breaking rule), but only if this gain is positive. If no positive gain is possible, i stays in its original community. This process is applied repeatedly and sequentially

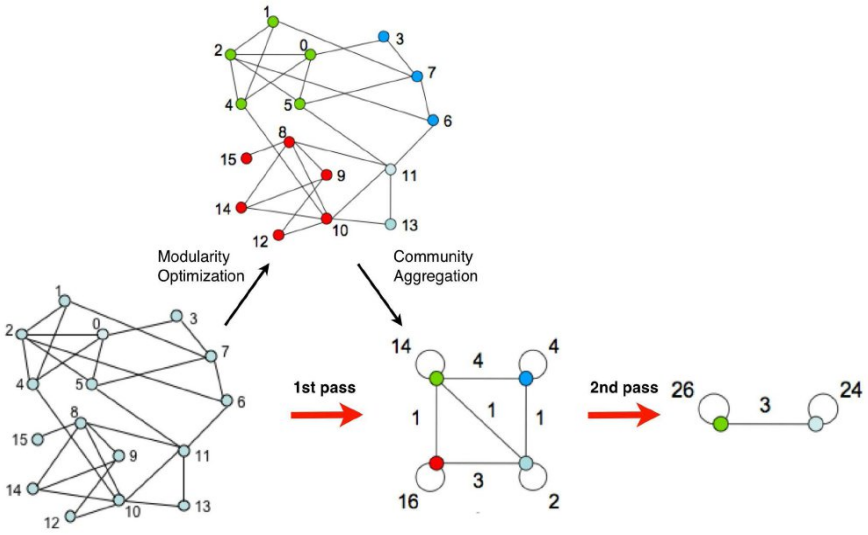

Figure 4: Visualization of the steps of the Louvain method algorithm. Each pass is made of two phases: one where modularity is optimized by allowing only local changes of communities; one where the found communities are aggregated in order to build a new network of communities. The passes are repeated iteratively until no increase of modularity is possible.

for all nodes until no further improvement can be achieved and the first phase is then complete. This first phase stops when a local maxima of the modularity is attained.

The second phase of the algorithm consists in building a new network whose nodes are now the communities found during the first phase. To do so, the weights of the links between the new nodes are given by the sum of the weight of the links between nodes in the corresponding two communities. Links between nodes of the same community lead to self-loops for this community in the new network. Once this second phase is completed, it is then possible to reapply the first phase of the algorithm to the resulting weighted network and to iterate. Let us denote by "pass" a combination of these two phases. By construction, the number of meta-communities decreases at each pass, and as a consequence most of the computing time is used in the first pass. The passes are iterated (see figure 4) until there are no more changes and a maximum of modularity is attained.

Analysis

One can directly observe that this algorithm is an instance of an agglomerative hierarchical clustering method, in the terms that we defined it; in this case, the agglomerative criterion is defined as being the gain of modularity of merging two clusters (i.e. by removing a node i from its community and by placing it in another community of some node j). Yet the major differences from classical agglomerative methods lay in two observations:

the richness of the search process and the hierarchical output obtained.

For the search complete every combination of nodes (each leading to a particular partitioning in a set of communities) should be tested in order to find a solution which is globally optimal in terms of modularity; the problem is clearly NP-hard. Since this solution is not practicable, the search space must be reduced with, of course, the inevitable drawback of losing completeness.

Recall that in classical hierarchical clustering, communities are successively merged two-by-two, each merging locally optimizing the objective function. When two communities are merged together, *they can then no longer be separated*, and the next locally optimal merging is searched. Fixing the communities after each merging considerably reduces the search space.

During the first phase of the Louvain method algorithm, nodes are moved between incident communities; different arrangements of nodes are successively found, each increasing the modularity until a local optima is found. In the second phase, the communities provided by the final arrangement of nodes are fixed when communities are transformed into the new set of nodes. The *merging of two communities is thus final only after convergence of this search process (first phase) to a local optima* (i.e. after a sequence of merging in which the association of a given community is not necessarily definitive), leading to a partial solution that can involves combination of merged communities that could not be selected in a classical hierarchical fashion, in which each merging between two communities is definitive. The search strategy applied in the Louvain method is then richer than the classical methods (as defined above), allowing potentially better solutions to be reached.

We thus highlighted the fact that the Louvain method is more efficient, in terms of the modularity obtained, than a classical approach probably because the search space explored is richer, bigger than classical ones. What about the computational efficiency of the method? One could assert that the richer the search space, the more the algorithm will spend time to explore it (and is likely to spend more time before reaching local optimae); this is of course true in a naive approach but in this case, we can do better. In fact, the algorithm has the property of *incrementality*. Deville defines the property of incrementality in [1] as follows: intuitively, an algorithm is incremental when the computation of a new result from a previous result and with a *modification* of the previous input is more efficient than the computation of the new result from the new input. The Louvain method has this property, because Blondel *et al* actually provide an expression to efficiently compute the gain in modularity when moving an isolated node i into a community C :

$$\Delta Q = \left[\frac{\sum_{in}+k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \qquad (2)$$

, where $\sum_{in}$ is the sum of the weights of the links inside C, $\sum_{tot}$ is the sum of the weights of the links incident to nodes in C, k_i is the sum of the weights of the links incident to node i, $k_{i,in}$ is the sum of the weights of the links from i to nodes in C and m is the sum of the weights of all the links in the network. Equation 2 thus provides a

way to efficiently compute the new modularity (new result) after a modification (of the previous input), given that the modularity before the modification is known (i.e. from the previous result). The naive and far less efficient way being of course to recompute the whole modularity of the graph using equation 1.

Let us now focus on the hierarchical output obtained by the Louvain method in comparison with a classical approach. With the Louvain method algorithm, each "pass" of the whole process gives a higher-level hierarchical output; the height of the hierarchy constructed is then determined by the number of passes, which is generally a small number. In the classical approach, each merging produces a new hierarchical level, producing thus $n-1$ different levels (where n is the initial number of nodes). Although the output provided by a classical approach is richer than the one provided by the Louvain method, the later is however more concise and generally more precise, by focusing on more important transitions in the number of communities.

Search improvements

In [9], several solutions have been suggested to increase the efficiency of the algorithm.

A partial optimization method consists of stopping the algorithm (within a pass and between two passes) whenever the modularity gain is below a given threshold.

The method that we have tested is the leaves removal of an unweighted and simple graph (no self-loops): before the first pass, we remove the nodes of degree 1 and we add a self-loop to their neighbors. We reinsert the nodes at the end into their neighbors' community. The resulting algorithm is a bit faster for the Internet dataset whereas it is way slower than the original version for the other datasets since the required preprocessing is important (it really depends on the number of edges).

Another heuristic suggested is whether or not considering the move of the nodes. Indeed, less and less nodes are moved from one community to another one. Because of considering the move of a node has a certain cost, one would like to freeze a node that has not moved since the last iteration for one iteration, or only consider it if its neighborhood has changed, or according to a given probability...

At last, another suggested optimization is introducing simulated annealing for the communities' selection: instead of picking the community whose modularity gain is maximum, one would pick another one with a decreasing probability.

Test results

Figure 5 shows the test results obtained on a dual core 2.26GHz with 3Gb of ram using Python 3.2.3 , on the Karate [10], Arxiv [11], hep-th-citations and hep-ph-citations [11] and Internet [12] test sets. We compare the results obtained on the same machine, using the C++ implementation used in [3], our Python implementation, and finally our implementation with leaves removal heuristic. As the table reads, the heuristic we implemented badly performs on the different datasets. This is mainly due to the

	Karate	Arxiv	hep-th-citations	hep-ph-citations	Internet
Nodes / Links	34 / 77	9377 / 38214	27770 / 352807	34546 / 421578	22963 / 48436
C++	0.002s	0.111s	0.727s	1.311s	0.179s
PyLouvain	0.071s	15.863s	3m44.365s	5m47.029s	2m14.431s
PyLouvain with leaves removal	0.072s	15.962s	5m18.037s	7m25.481s	1m59.634s

Figure 5: Results obtained on some classical test instances. Time measures include pre-processing of the graph description file (about 30 seconds for Internet).

important computing time needed to remove the leaves[6], i.e. the nodes having a degree of 1. This computing time is prohibitive since the percentage of leaves was not that important.

5 Conclusion

Community detection (and clustering in a more general purpose) is a wide and active subject of research, and applications are large. Whilst the algorithm we saw in this report mainly relies on the measure of modularity, we also noticed that other measures exist and thus one should always wonder whether the modularity is the most appropriate measure of quality for a given clustering application and, otherwise should seek for a measure whose provides a better representation of the implicit objective function he wants to optimize.

However, we spotted the fact that the Louvain method is actually part of a particular class of clustering methods and provided it an analysis in terms of the size of the search space. Whereafter we briefly presented some heuristic improvements to the Louvain method and chose one which we implemented and tested; discussing deeply the different heuristics proposed in [9], testing or even thinking at new heuristics would be very interesting, but unfortunately we were constrained by the time we could allow to this project.

[6] Indeed, we use a side data structure that stores the edges of each node. Hence, removing a given node implies removing all its references. Note also that our algorithm is not the most optimized one could find: it actually aims at providing us a working prototype a first approach to the Louvain method.

Appendix : PyLouvain API

Our PyLouvain module is self-standing, that is, it does not require another module to run. All you need to run it is a working Python3 environment. There are three ways to feed a graph to PyLouvain:

```
pyl = PyLouvain(nodes, edges)
```

where nodes is a list of integers and edges is a list of ((node, node), weight) pairs;

```
pyl = PyLouvain.from_file(path)
```

where `path` is the string of a path (relative or absolute) to a text file containing "node node [weight]" lines (the weight is optional);

```
pyl = PyLouvain.from_gml_file(path)
```

where path is the string of a path (relative or absolute) to a text file compliant to the GML format [13]. To launch the algorithm, just issue the following line:

```
partition, modularity = pyl.apply_method()
```

the result is a (partition, modularity) pair.

References

[1] M. Girvan and M.E.J. Newman. *Community structure in social and biological networks*. Proc. Natl. Acad. Sci. USA 99, 7821-7826, 2002.

[2] M.E.J. Newman. *Modularity and community structure in networks*. Proc. Natl. Acad. Sci. USA 103, 8577-8582, 2006.

[3] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre, *Fast unfolding of communities in large networks*, Journal of Statistical Mechanics: Theory and Experiment, 1742-5468, P10008, 2008.

[4] Guimera R, Sales M and Amaral L A N, 2004 Phys. Rev. E 70 025101.

[5] Newman, M. E. J. (2004) Phys. Rev. E 69, 066133.

[6] Duch, J. & Arenas, A. (2005) Phys. Rev. E 72, 027104.

[7] Saerens M. *Data Mining and Decision Making*. UCL course LSINF2275.

[8] Deville Y. *Constraint Programming*. UCL course LINGI2365.

[9] Thomas Aynaud, Vincent D. Blondel, Jean-Loup Guillaume et Renaud Lambiotte, *Optimisation locale multi-niveaux de la modularité* (in French). Chapter 14 in: *Partitionnement de graphe Optimisation et applications*, Charles-Edmond Bichot et Patrick Siarry (Eds), Hermes, Paris, 2010.

[10] Zachary WW, 1977 Journal of Anthropological Research 33 452.

[11] http://www.cs.cornell.edu/projects/kddcup/ (Cornell KDD Cup)

[12] http://www-personal.umich.edu/~mejn/netdata/ (A symmetrized snapshot of the structure of the Internet at the level of autonomous systems, reconstructed from BGP tables posted by the University of Oregon Route Views Project.)

[13] http://www.fim.uni-passau.de/en/fim/faculty/chairs/theoretische-informatik/projects.html - Graph Modelling Language