Methods for community detection in multi-layer networks

Surervisor: Prof. Francesco Rinaldi
University of Padua

Co-surervisor: Prof. Francesco Tudisco
Gran Sasso Science Institute

Student: Sara Venturini
N. 1206680

17 July 2020
Academic year 2019/2020
Introduction

Complex network theory is an evolution of graph theory, whose origin traces back to Euler’s famous publication of 1736 on the “Seven Bridges of Königsberg” [1]. Since then a lot has been learned about graphs and their mathematical properties. Networks have emerged as effective tools for modelling and analysing complex systems of interacting entities. Graphs arise naturally in many disciplines, such as social networks (which arise via offline and/or online interactions) [2], information networks (i.e. hyperlinks between webpages on the World Wide Web) [3], infrastructure networks (i.e. transportation routes between cities) [4], and biological networks (i.e. metabolic interactions between cells or proteins, food webs) [5].

One of the most relevant issue of graphs representing real systems is the identification of communities, or clustering, i.e. the organization of vertices in clusters or modules, with more edges connecting vertices of the same group and fewer edges joining vertices of different groups. Community detection in large networks is potentially very useful. Nodes belonging to the same community might have other properties in common. Society offers a variety of possible group organizations: families, working and friendship circles, villages, towns, nations. In recent years, the spread of Internet has led to the development of virtual communities, which live on the Web. Social communities have been studied for a long time [2], but they also occur in many networked systems from biology, computer science, engineering, economics, etc. For example, biological systems are organized hierarchically with networks of communities interacting at various levels, from ecosystems to networks of synaptic connections between neurons, up to gene and metabolic networks. In protein-protein interaction networks, communities correspond to group proteins having the same specific function within the cell [5], in the graph of the World Wide Web they may represent groups of pages dealing with the same topics [6], in metabolic networks they may be related to functional modules [7]. Communities can have concrete applications. For example, clustering Web clients who have similar interests and are geographically close could be served by a dedicated server, improving the performance of services provided on the World Wide Web [3]; or clusters of large graphs can be used to create data structures and handle navigational queries, like path searches [8].

Community detection is important also to classify vertices, according to their structural position in the modules. So, vertices with a central position in their clusters, i.e. sharing many edges with the other nodes of the group, may have an important function of control and stability within the group; vertices lying at the boundaries between clusters are important for leading
the relationships between communities.

Another important aspect related to community structure is the hierarchical organization. Real networks are usually composed by communities including smaller communities, which in turn include smaller communities, etc. For example, the human body is composed by organs, organs by tissues, tissues by cells, and so on.

Community detection is a very hard problem and not yet satisfactorily solved, despite the extensive studies in the literature. Many algorithms have been developed, however they are designed for single-layer networks. This assumption is almost always a simplification, which can lead to misleading results. Recent advances in the study of networked systems have shown that the interconnected world is composed of networks that are coupled to one another through different layers, where each layer represents one of many possible types of interactions. For example, individuals have multiple relationships in social networks, such as economic, political, and financial. In biology, protein interaction networks consist of seven distinct layers that account for different genetic and physical interactions. Analysing multi-layer networks is of great importance because many interesting patterns cannot be obtained by analysing single-layer networks. For example, in multi-layer cancer networks where each layer corresponds to a specific clinical stage, a community represents a biological pathway that is critical for cancer diagnosis and therapy. However, it is hard to extract communities in multi-layer networks because of two reasons. First, multi-layer communities cannot be easily quantified because analyses on these multi-layer networks remain lacking. Second, complexity of multiple networks poses a challenge on finding algorithms for identifying communities in the multi-layer case. Despite these difficulties, great efforts have been devoted to the extraction of multi-layer communities. Most of the current algorithms either reduce multi-layer networks into a single-layer network or extend the algorithms for single-layer networks by using consensus clustering. However, these algorithms are criticized for their low accuracy because they either cannot preserve the community structure in compressed networks or ignore the connection among various layers. To overcome these problems, we must simultaneously take into account multiple layers.

In this thesis project, we propose multiple algorithms for community detection in multi-layer networks. The algorithms are based on the popular Louvain heuristic method for single-layer networks [9], which is a locally greedy modularity-increasing sampling process over the set of partitions.

The work is organized as follows.

Chapter 1 presents a broad description of the community detection problem. Firstly, we introduce some primary notions of graph theory and then we focus on the main problems that arise from community detection.

Chapter 2 contains a summary of the most relevant methods related to the study of community detection. In the first Section we focus on methods for single-layer graphs. In the second Section we study algorithms proposed for the multi-layer case.
Chapter 3 is the main contribution of this thesis and it presents our algorithms in detail. The first algorithm is called Louvain extension and the second one is named Louvain Multiobjective. Both of them try to extend the Louvain heuristic method to the multi-layer case. We present different variants of these two algorithms.

Chapter 4 shows some results obtained testing the methods. We did some tests on both artificial and real world networks to better compare the performances of the algorithms.

Finally, Chapter 5 reports some conclusions taking into consideration both the formulation and the experimentation of the methods.
# Contents

1 Problem Statement  

2 Related Work  
   2.1 Community Detection in Single-Layer Graphs  
   2.1.1 Traditional clustering methods  
   2.1.2 Modern clustering methods  
   2.2 Community Detection in Multi-Layer Graphs  
   2.2.1 Community detection in two-layer graphs  
   2.2.2 Community detection in multi-layer graphs  

3 Presentation of the Methods  
   3.1 Louvain Expansion  
   3.2 Louvain Multiobjective  

4 Experimentation  
   4.1 Evaluation  
   4.2 Artificial Networks  
      4.2.1 Informative case  
      4.2.2 Noisy case  
   4.3 Real World Networks  
      4.3.1 Informative case  
      4.3.2 Noisy case  

5 Conclusions  

Appendix  
   A Matlab code of the community-average method (Section 3.1)  
   B Matlab code of the community-variance-minus method (Section 3.1)  
   C Matlab code of the community-variance-plus method (Section 3.1)  
   D Matlab code of the multi-average method (Section 3.2)
E  Matlab code of the multi-variance-minus method (Section 3.2)  111
F  Matlab code of the multi-variance-plus method (Section 3.2)  123
Bibliography  135
Chapter 1

Problem Statement

Graph theory is extremely useful as representation of a wide variety of systems in different areas, such as biology [5], sociology [2], technology [3], and many others. All these networks can be studied as graphs, thus graph analysis has become crucial to understand the features of these systems.

A graph $G$ is composed by a pair of sets $(N,E)$ where $N$ is the set of nodes and $E$ is the set of edges and it is a subset of all the possible pairs of nodes in $V$.

The adjacent matrix of a graph is a square $|N| \times |N|$ matrix $A$ such that its element $A_{ij}$ is one when there is an edge from vertex $i$ to vertex $j$, and zero otherwise.

In this work we only consider undirected graphs and just a single edge between nodes is allowed.

Many complex systems are composed of coupled networks through different layers, rather than just one, where each layer represents one of many possible types of interactions. They are called multi-layer networks.

We represent a multi-layer graph with $k$ layers through a sequence $(N_s,E_s)_{s=1,...,k}$ where $N_s$ and $E_s$ are respectively the set of nodes and the set of edges of layer $s$.

Connected to a multi-layer graph, we can consider a set of adjacent matrices $(A_s)_{s=1,...,k}$ where $k$ is the number of layers and $A_s$ is the adjacent matrix of layer $s$.

We focus our attention on multi-layer graphs where just edges vary between layers, thus each node is present in all layers, and there are not edges between nodes of different layers.

Community detection has attracted significant attention during the recent years. The goal of community detection is to partition vertices of a graph into densely-connected components (i.e. the so called communities).

The main problems of graph clustering are the concepts of community and partition, which are not rigorously defined, although they are intuitive concepts.

Intuitively, we get the notion that a community must have more edges between the nodes of the community than edges linking nodes of the community with the rest of the graph. However, there are many alternative definitions of community.
Local definitions consider communities as separate entities and evaluate them independently of the rest of the graph. Global definitions, despite the local ones, consider communities as an integral part of the graph, which cannot be studied regardless of the rest. There are many global criteria to find communities, some of them use some global properties just indirectly, incorporating them in an algorithm that shows communities at the end. Some definitions are based on vertex similarity. The main idea is that a community is a group of nodes similar to each other, so they compute the similarity between each pair of nodes and put most similar nodes in the same community.

Another problem of community detection is the concept of partition. A partition is a division of a graph in clusters, such that each node belongs to just one of them. In real systems we can have overlapping communities, where a node can belong to more communities. However, we don’t consider this case in our work. The main problem is to distinguish a good partition from a bad one. For this reason, we need a quantitative criterion to measure the goodness of a graph clustering. A quality function is a function that assigns a number to each partition of a graph, thus one can sort them by their value and identify the best one. Nevertheless, the answer depends on the quality function and on the community concept that are used. This problem of quantifying the value of partitions becomes even more complicated in the multi-layer case, because a given partition can be very good for one layer but very bad for another.

In this work we rely on the idea that a graph has a community structure if it is different from a random graph. In fact, a random graph should not have a community structure, since any two nodes have the same probability to be adjacent or not. A null model is used as term of comparison, to verify if whether a graph shows a community structure or not. For this reason, it maintains some structural features of the original graph. This concept is the basis of the definition of modularity, a quality function where a subgraph is a community if the number of edges inside the subgraph exceeds the expected number of internal edges that the same subgraph would have in the null model. Modularity can be written as follows

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - P_{ij}) \delta(C_i, C_j)$$

where the sum runs over all pairs of vertices, $A$ is the adjacency matrix, $m$ the total number of edges of the graph, $P_{ij}$ represents the expected number of edges between vertices $i$ and $j$ in the null model, and $C_i$ is the community of node $i$. The function $\delta$ yields one if vertices $i$ and $j$ are in the same community ($C_i=C_j$), zero otherwise.

In the literature have been suggested several versions of modularity and null model. The most popular null model has been proposed by Newman and Girvan, where edges are linked at random, under the constraint that the expected degree of each vertex of the null model stays the same as the degree of the correspondent node in the original graph [10]. So, the final expression of modularity reads

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

(1.2)
where $k_i$ is the degree of node $i$ (the sum of all the edges incident to $i$).

Large positive values of modularity indicate good partitions. The modularity of the whole graph, taken as a single community, is zero, and it is always smaller than one, and can be negative as well. For instance, the partition in which each vertex is a community is always negative. This implies that, if there are no partitions with positive modularity, the graph has no community structure. Another feature of modularity is that the gain due to the movement of a node from one community to another can be easily calculated.

Unfortunately, modularity has got some limits. The main problem is the so-called resolution limit [11], that may prevent to find community that are small with respect to the graph as a whole, even if they are cliques. Thus, when the partition with maximum modularity includes clusters with total degree of the order of $\sqrt{m}$ or smaller, where $m$ is the number of total edges of the graph, one cannot distinguish if the clusters are single communities or if they are combined together. The reason of the resolution limit is the definition of the null model, where we assume that each node can interact with every other node, but more reasonably each node interacts just with a part of the graph. This limit has a large impact in real applications.

To overcome this problem, one could recursively apply modularity optimization on each community of the obtained clustering [11] [12]. However, we don’t have a stopping criteria and each cluster uses a different null model, since communities can be of different sizes.

*Good et al.* [13] discovered that modularity values are very close to the global maximum, nevertheless partitions with high modularity are not necessarily similar to each other and the global maximum is impossible to reach.

Another disadvantage of modularity is that can be applied just to single-layer graphs. In this thesis project we hence propose some models that take advantage of this modularity function and try to extend it to the multi-layer case. All the models are based on the Louvain heuristic method for single-layer networks [9], which is a locally greedy modularity-increasing sampling process over the set of partitions. The most intuitive idea, already studied in the literature, is to use the modularity average on the layers. Thus, we propose two methods that take into account variance of modularity on the layers, in addition to the average. We present also a more sophisticated filter type algorithm. We focus on the multiobjective aspect of the problem and maintain just the modularity vectors that are not dominated according to a suitably developed Pareto search. We implemented these methods in Matlab and we performed some experiments both on artificial and real world networks.
Chapter 2

Related Work

In this Chapter, we present a brief review of the literature related to the community detection problem. In the first Section 2.1, we start by describing the classical approaches for single-layer graphs, following Fortunato [14]. *Traditional algorithms* include graph partitioning, hierarchical, partitional and spectral clustering; *modern methods* include divisive algorithms, modularity based methods, spectral algorithms, dynamic algorithms, methods based on statistical inference, non-negative matrix factorization, nonlinear spectral algorithms and total variation approaches. In Section 2.2 we present some algorithms for multi-layer graphs: first we introduce community detection algorithms in two-layer graphs, later detection algorithms that can support multi-layer graphs containing more than or equal to two layers.

2.1 Community Detection in Single-Layer Graphs

Many community detection approaches have been proposed for single-layer graphs. Fortunato conducted a survey on this topic [14]. He presents:

- *traditional clustering methods* divided into: graph partitioning, hierarchical, partitional and spectral clustering;

- *modern methods*, divided into categories based on the type of approach: divisive algorithms, modularity based methods, spectral algorithms, dynamic algorithms, methods based on statistical inference. In this section we add some more recent methods: non-negative matrix factorization, nonlinear spectral algorithms and total variation approaches.

We do not present algorithms to find overlapping communities, multiresolution and hierarchical techniques because they are not connected to our work.

2.1.1 Traditional clustering methods

Traditional clustering methods include graph partitioning, hierarchical, partitional and spectral clustering.
Graph partitioning

Graph partitioning aims to divide the vertices in an established number of groups of predefined size, such that the cut size, i.e. the number of edges running between clusters, is minimal. Specifying the number of clusters of the partition is necessary. The graph partitioning problem is NP-hard, however several algorithms have good results, even if their solutions are not necessarily optimal [15]. Many algorithms perform a bisection of the graph, and partitions into more than two clusters are usually attained by iterative bisectioning [16]. The well-known max-flow min-cut theorem by Ford and Fulkerson [17], that states that the minimum cut between any two vertices \( s \) and \( t \) of a graph carries the maximum flow that can be transported from \( s \) to \( t \) across the graph, has been used to determine minimal cuts from maximal flows in clustering algorithms [18] [19]. We usually do not have information about the community structure of a graph, in such cases this procedure is not useful, and one must make some unjustified assumptions about the number and size of the clusters.

Hierarchical clustering

The vertices of a graph can be grouped at different levels, with small clusters included within large clusters, which are in turn included in larger clusters, and so on. In order to reveal the multilevel structure of the graph, one may use hierarchical clustering algorithms [20]. Hierarchical clustering is very common in social network analysis, biology, engineering, marketing. As the base of hierarchical clustering methods there is the definition of a similarity measure between vertices. After a measure is chosen, one computes the similarity for each pair of vertices, obtaining the similarity matrix. Hierarchical clustering techniques can be classified in two categories: agglomerative algorithms, in which clusters are iteratively merged if their similarity is sufficiently large, and divisive algorithms, in which clusters are iteratively split by removing edges connecting vertices with low similarity. Hierarchical clustering has the advantage that a preliminary knowledge on the number and size of the clusters is not required. However, it does not give us a way to discriminate between the many partitions obtained by the procedure.

Partitional clustering

Partitional clustering is another common class of methods to find communities in a graph. The number of clusters is preassigned, say \( k \). Each vertex is a point of a metric space and a distance measure is defined between them. The distance is a measure of dissimilarity between vertices. The purpose is to partition the points in \( k \) clusters so to maximize/minimize a given cost function based on distances between points. The most popular partitional technique in the literature is \( k \)-means clustering [21], where the cost function is the total intra-cluster distance, or squared error function. The limitations of partitional clustering is both that the number of clusters must be specified at the beginning, as for the graph partitioning algorithms, and that the embedding in a metric space for some graphs can be not natural.
Spectral clustering

Supposed to have a pairwise similarity function $S$ defined between a set of $n$ objects, which is symmetric and non-negative. Spectral clustering includes all methods and techniques that partition the set into clusters by using the eigenvectors of matrices, like $S$ itself or other matrices derived from it. The objects could be points in some metric space, or the vertices of a graph. Spectral clustering consists of a transformation of the initial set of objects into a set of points in space, whose coordinates are elements of eigenvectors: the set of points is then clustered via standard partitional techniques, like k-means clustering. The first contribution on spectral clustering was a paper by Donath and Hoffmann [22], who used the eigenvectors of the adjacency matrix for graph partitions. In the same year, Fiedler [23] used the eigenvectors of the Laplacian matrix, by far the most used matrix in spectral clustering.

2.1.2 Modern clustering methods

Modern clustering methods include divisive algorithms, modularity based methods, spectral algorithms, dynamic algorithms, methods based on statistical inference, non-negative matrix factorization, nonlinear spectral algorithms and total variation approaches.

Divisive algorithms

The problem of divisive algorithms consists in detecting the edges that connect vertices of different communities and remove them, so that the clusters get disconnected from each other. The critical point is to find a property of intercommunity edges that could identify them. Divisive methods just perform hierarchical clustering on the graph at study, so they do not introduce new techniques. The most popular algorithm is that proposed by Girvan and Newman [24] [10]. Edges are selected according to the values of measures of edge centrality, estimating the importance of edges according to some property or process running on the graph. Girvan and Newman study in deep the concept of betweenness, which is a variable expressing the frequency of the participation of edges to a process. Many modifications of this method have been proposed, like the algorithm by Holme et al. where vertices, rather than edges, are removed [25]; the algorithm proposed by Pinney and Westhead, that is able to find overlapping communities [26]; the method designed by Estrada based on the concept of communicability between nodes [27].

Modularity based methods

Newman-Girvan modularity [28], originally introduced to define a stopping criterion for the algorithm of Girvan and Newman, has rapidly become the most used quality function. Modularity optimization is an NP-complete problem, however there are currently several algorithms able to find good approximations of the modularity maximum in a reasonable time. We concentrate on clustering methods that require modularity, directly and/or indirectly: greedy techniques, simulated annealing, extremal optimization, spectral optimization. The first greedy technique to maximize modularity was designed by Newman [29]. It is an agglomerative hierarchical clustering method, where groups of vertices are successively joined to
form larger communities such that modularity increases after the merging. At the beginning, all vertices of the graph are put in different communities. Edges are not initially present, they are added one by one during the procedure. An edge is chosen such that this partition gives the maximum increase of modularity with respect to the previous configuration. A lot of modifications of this method have been proposed [30], [31], [32].

A different greedy approach is the Louvain method, introduced by Blondel et al. [9]. Initially, each vertices is a community. The first step consists of a sequential sweep over all vertices. Given a vertex $i$, one computes the gain in weighted modularity coming from putting $i$ in the community of its neighbour $j$ and picks the community of the neighbour that yields the largest increase of modularity, as long as it is positive. At the end of the sweep, one obtains the first level partition. In the second step communities are replaced by supervertices. Two supervertices are connected if there is at least an edge between vertices of the corresponding communities and the weight of the edge between the supervertices is the sum of the weights of the edges between the represented communities at the lower level. The whole procedure is repeated iteratively. At some point, the algorithm stops because modularity cannot increase anymore.

Simulated annealing [33] is a probabilistic procedure for global optimization used in different fields and problems. It performs an exploration of the space of possible states, looking for the global optimum of a function $F$, say its maximum. Transitions from one state to another occur with probability 1 if $F$ increases after the change, otherwise they occur with a probability $\exp(\beta \Delta F)$, where $\Delta F$ is the decrease of the function and $\beta$ is an index of stochastic noise, which increases after each iteration. Simulated annealing was first used for modularity optimization by Guimerà et al. [34]. Its standard implementation [7] is composed by two types of moves: local moves, where a single vertex is shifted from one cluster to another randomly; global moves, consisting of mergers and splits of communities. The method can potentially come very close to the true modularity maximum, but it is slow.

Extremal optimization (EO) is a heuristic search procedure proposed by Boettcher and Percus [35], in order to achieve an accuracy comparable with simulated annealing, but with a gain in computer time. It is based on the optimization of local variables, expressing the contribution of each unit of the system to the global function at study. Duch and Arenas [36] used this technique for modularity optimization. Modularity can be written as a sum over the vertices: the local modularity of a vertex is the value of the corresponding term in this sum. Dividing the local modularity of the vertex by its degree, we obtain a fitness measure for each vertex. In this way the measure is normalized and does not depend on the degree of the vertex. At the beginning, the vertices are divided randomly into two groups of the same size. At each iteration, the vertex with the lowest fitness is shifted to the other cluster. The move changes the partition, so the local fitnesses of many vertices need to be recalculated. The process stops when the global modularity cannot be improved any more. After the bipartition, each cluster is considered as a graph on its own and the procedure is repeated, as long as the global modularity increases for the partitions found.

Spectral optimization aims to optimize modularity using the eigenvalues and eigenvectors of the modularity matrix. One can maximize modularity via spectral clustering, by replacing the Laplacian matrix with the modularity matrix [10] [29].

In the most recent literature on graph clustering, several modifications and extensions of modu-
larity can be found, motivated by specific classes of clustering problems or graphs that one may want to analyse. For instance, due to the so-called resolution limit of the modularity function, Traag at al. compared the network to a constant factor, instead of a random null model as usual [37].

Spectral algorithms

Spectral properties are used to find partitions, as in the case of spectral clustering, which considers the eigenvectors of Laplacian matrix, or in the optimization of modularity, which uses the eigenvectors of the modularity matrix.

The algorithm proposed by Donetti and Muñoz uses the eigenvectors of the Laplacian matrix [38]. It turns vertices into points of a metric space, using the eigenvectors components as coordinates, because they are close for vertices in the same community.

Also the algorithm designed by Alves uses spectral properties of the Laplacian matrix [39]. Here effective conductances for pairs of vertices is computed, looking at the graph as an electric network with edges of unit resistance.

Capocci et al. used the eigenvectors of a right stochastic matrix, that should have similar properties as the Laplacian [40].

Yang and Liu proposed a recursive bisectioning procedure, which uses the spectral properties of the adjacency matrix [41].

In recent times, Fasino and Tudisco studied in deep spectral properties of modularity matrices, that are related to the community detection problem [42] [43].

Dynamic algorithms

Dynamic methods use processes running on the graph, focusing on spin-spin interactions, random walks and synchronization.

One of most popular spin model in statistical mechanics, is the Potts model [44]. It describes a system of spins that can be in $q$ different states. If Potts spin variables are assigned to the vertices of a graph with community structure, and the interactions are between neighbouring spins, the modules could be likely recovered from like-valued spin clusters of the system, as there are many more interactions inside communities than outside.

Random walks [45] can also be useful to find communities. If a graph has a strong community structure, a random walker spends a long time inside a community due to the high density of internal edges and consequent number of paths that could be followed.

Synchronization [46] is an emergent phenomenon occurring in systems of interacting units and is present in nature, society and technology. In a synchronized state, the units of the system are always in the same or similar state. Synchronization has also been applied to find communities in graphs. If oscillators are placed at the vertices, with initial random phases, and have nearest-neighbour interactions, oscillators in the same community synchronize first, whereas a full synchronization requires a longer time. So, if one follows the time evolution of the process, states with synchronized clusters of vertices can be stable and durable, so they can be easily individuated.
Methods based on statistical inference

Statistical inference [47] has the intent to deduce properties of data sets, starting from a set of observations and model hypotheses. If the data set is a graph, the model, based on hypotheses about connections between nodes, has to fit the actual graph topology. These methods try to find the best fit of a model to the graph, where the model assumes that vertices have some sort of classification, based on their connectivity patterns.

Generative models adopt Bayesian inference [48], in which the best fit is obtained through the maximization of a likelihood. Bayesian inference uses observations to estimate the probability that a given hypothesis is true. It is based on two elements: the evidence, expressed by the information one has about the system (e.g. through measurements), and a statistical model with parameters.

Block modelling [49] is a popular approach in statistics and social network analysis to decompose a graph in classes of vertices with common properties. Vertices are grouped in classes of equivalence. There are two main definitions of topological equivalence for vertices: structural equivalence [50], in which vertices are equivalent if they have the same neighbours; regular equivalence [51] [52], in which vertices of a class have similar connection patterns to vertices of the other classes.

Model selection [53] try to find models which are both simple and good at describing a system/process. To select a model, there is not one defined way, but some heuristics.

Information theory has also been used to detect communities in graphs. Ziv et al. [54] have designed a method in which the information contained in the graph topology is compressed such to preserve some predefined information. This is the underlying philosophy of the information bottleneck method [55].

Non-negative Matrix Factorization

Non-negative Matrix Factorization (NMF) [56] [57] has been applied to many applications such as clustering and classification. It provides a linear representation of non-negative data in high dimensional space with the product of two non-negative matrices. Some papers explicitly include the notion of sparseness, improving the found decompositions [58] [59]; some others incorporate discriminant constraints in the decomposition [60]. Some recent research works suggest that data of many applications in a high dimensional Euclidean space are usually embedded in a low dimensional manifold [61]. To explore the local structure on the low dimensional manifold, papers have proposed Locality Preserving NMF and Neighbourhood Preserving NMF, which add constraints between a point and its neighbours [62] [63].

Nonlinear Spectral algorithms and Total Variation approaches

Modularity optimization is an NP-complete problem. We have seen that many algorithms, such as spectral clustering methods or non-negative matrix factorization (NMF) methods, relax the discrete optimization space into a continuous one to obtain an easier optimization procedure. However, in general the solution of the relaxed continuous problem and that of the discrete NP-hard problem can be very different. A new set of algorithms obtains tighter relaxations, taking
idea from the image processing literature. They are based on the concept of total variation (TV), which favours the formation of sharp indicator functions in the continuous relaxation. These functions equal one on a subset of the graph, zero elsewhere and exhibit a non-smooth jump between these two regions. At the beginning, total variation techniques had a recursive bi-partitioning procedure to handle more than two classes. Later, Bühler and Hein and Bresson et al. proposed two methods that do not rely on a recursive procedure [64] [65].

In 2013, Hu et al. showed that modularity optimization is equivalent to minimizing a particular non-convex total variation based functional over a discrete domain [66] and, in 2018, Boyd et al. showed that this equivalence states for a convex total variation based functional [67]. Both algorithms assume that the number of communities is known. This process is called non-linear exact relaxation of the modularity function.

In recent times, Tudisco et al. and Tudisco and Higham proposed instead nonlinear spectral methods [68] [69]. Finally, in 2020, Cristofari et al. introduced the modularity total variation (TVQ) and showed that its box-constrained global maximum coincides with the maximum of the original discrete modularity function [70].

2.2 Community Detection in Multi-Layer Graphs

In contrast to the community detection problem in single graphs, new challenges arise for community detection in multi-layer graphs. It is natural to detect multi-layer communities by extending the algorithms for single-layer community detection. The most popular approaches that have been employed for this extension can be grouped into two strategies: the first one reduces the multi-layer networks into a single-layer network and then applies single-layer network algorithms to obtain the communities in the collapsed network [71]; whereas the second strategy obtains the communities for each layer applying single-layer network algorithms, and then combines the obtained communities by using consensus clustering [72]. However, these algorithms are criticized for their low accuracy because they either cannot preserve the community structure in compressed networks or ignore the connection among various layers. Another problem is connected to noise, in fact these methods usually suppose that each layer is informative but in real networks some of them are just noise. To overcome these problems, algorithms must simultaneously take into account multiple layers. Thus, there is necessity to develop effective algorithms for community detection in multi-layer networks, rather than by simply extending the available single-layer network algorithms.

In their work, Kim and Lee [73] divide community detection algorithms for two-layer graphs, from detection algorithms that can support multi-layer graphs containing more than or equal to two layers. We follow this subdivision in our review.

2.2.1 Community detection in two-layer graphs

In this subsection, we introduce community detection algorithms in two-layer graphs. All algorithms described in this section can only support two-layer graphs and mostly consider structural and attribute information.
Cluster Expansion

Li et al. [74] proposed a hierarchical community detection algorithm based on both relations and textual attributes using the cluster expansion idea. Initially the algorithm quickly finds the centers as seed of communities, then it expands the centers into the communities. The algorithm is composed by four steps: core probing, core merging, affiliation, and classification.

Matrix Factorization

Qi et al. [75] proposed a community detection algorithm based both on link structure and edge content using the Edge-Induced Matrix Factorization (EIMF). The key point of this algorithm is the use of edge content for the community detection process, which can be useful when nodes interact with multiple communities, since it can help distinguishing between the different interactions of nodes. This algorithm firstly takes into consideration just the link structure, then it incorporates also the edge content.

Unified Distance

Zhou et al. [76] proposed a community detection algorithm, called SA-Cluster, based on both structural and attribute similarities using a unified distance measure. The main contributions of SA-Cluster are a unified distance measure to take simultaneously into account both structural and attribute similarities and a weight self-adjustment method to control the degree of importance of structural and attribute similarities.

Model-Based Method

Xu et al. [77] proposed a model-based community detection method based on both structural and attribute aspects of a graph. The main idea of this approach is the use of a probabilistic model that fuses both structural and attribute information instead of an artificial distance measure. The algorithm firstly constructs the probabilistic model and then a variational approach to solve it.

Pattern Mining

Silva et al. [78] proposed a community detection algorithm based on structural correlation pattern mining, called SCPM. The key point of SCPM is to reveal the connection between vertex attributes and dense subgraphs using both frequent itemset mining and quasi-clique mining. Here, a dense subgraph is defined by a quasi-clique.

Graph Merging

Ruan et al. [79] proposed a community detection approach, called CODICIL, to combine structural and attribute information using the graph merging process. The main contribution of this algorithm is to delate noise in the link structure using content information.
2.2.2 Community detection in multi-layer graphs

In this section, we introduce community detection algorithms that can support multi-layer graphs containing more than or equal to two layers.

Matrix Factorization

Tang et al. [80] and Dong et al. [81] proposed graph clustering algorithms for multi-layer graphs based on matrix factorization. The key point of these two algorithms is to fuse different information by extracting common factors from multiple layers, which may then be used by general clustering methods. Tang et al. [80] approximates adjacency matrices while Dong et al. [81] approximates graph Laplacian matrices.

Pattern Mining

Zeng et al. [82] proposed a subgraph mining algorithm for finding quasi-cliques that appear on multiple layers with a frequency above a given threshold. The main contribution of this algorithm is to find cross-graph quasi-cliques that are frequent, coherent, and closed. Generally, the cross-graph quasi-clique has been defined as a set of vertices belonging to a quasi-clique that appears on all layers and must be the maximal set [83].

Spectral Methods

Some methods try to extend spectral clustering to multi-layer graphs. In general, these algorithms aim to define a graph operator that contains all the information of the multi-layer graph such that the eigenvectors corresponding to the smallest eigenvalues are informative about the clustering structure. These methods usually rely on some sort of arithmetic mean, for example the Laplacian of the average adjacency matrix or the average Laplacian matrix [84]. Further examples are the work of Zhou and Burges, which defines a multiple cut graph, which is good on average while it may not be the best on single graphs [85], and the algorithm designed by Chen and Hero, that performs convex aggregation of layers based on graph noise models [86].

Other Approaches

Some approaches adopt Bayesian inference [48], in which certain hypotheses about connections between nodes are made to find the best fit of a model to the graph through the optimization of a suitable likelihood [87].

Other methods try to propose an extension of Newman’s modularity [28] and connected null models. For instance, the method proposed by Wilson et al. finds overlapping communities and proves consistency in a suitable multi-layer stochastic block model [88]. A co-training approach is proposed by Kumar and Daumé [89], where the algorithm aims to find a consistent clustering under the main assumption that all the layers are informative, so each single layer has a piece of meaningful information from its own perspective. Kumar et al. [90] concentrated on this approach under the notion of co-regularization.
Chapter 3

Presentation of the Methods

In this Chapter, we present some methods for community detection in multi-layer networks. The algorithms are based on the Louvain heuristic method for single-layer graphs [9], that we mentioned in Chapter 2 describing modularity based methods.

The natural extension of this method to the multi-layer case, already studied in the literature, is to locally maximize the modularity average on the layers during phase 1, instead of the modularity of a single layer. In Section 3.1, we suggest two variants that, in the selection criteria of the algorithm during phase 1, take into account the variance of modularity on the layers, in addition to the average. This is due to the fact that in real networks we can have two cases: the informative case, where there are all informative layers, so each single layer has a piece of meaningful information from its own perspective, but we can also have the noisy case, where there are some noisy layers, which give us wrong information about communities.

In Section 3.2, we present a more sophisticated filter type algorithm. We are studying a multi-objective optimization problem, so we decide to memorize all the possible moves in a list and then we maintain just the modularity vectors that are not dominated according to a suitably developed Pareto search. We use different functions as criterion for handling the list, in particular, we use the average on modularity of the layers, and the convex combination of the average and the variance, thus getting two different approaches. Notice that these methods with unit length of the list correspond respectively to the algorithms described before.

These algorithms coincide with the Louvain method on single layer graphs.

Matlab codes of the methods are available in the Appendix.

3.1 Louvain Expansion

In this Section, we present the Louvain Expansion method for community detection in multi-layer networks. It is based on the Louvain method for single-layer graphs [9] and tries to expand it to the multi-layer case.

The input of the algorithm is a multi-layer graph $G$ with $k$ layers. We suppose to have multi-layer graphs where just edges vary between layers, thus each node is present in all layers, and
there are not edges between nodes of different layers. Each layer is an undirected graph and just a single edge between nodes is allowed. The output of the algorithm is a final assignment of nodes to communities.

The algorithm is composed of two phases that are repeated iteratively. At the beginning of the first phase, each node forms a community. The algorithm calculates some values related to this first partition, s.a. $Q_s$ the modularity of the clustering in layer $s$ for $s=1,...,k$ and a defined function $F$ connected to them. We consider as neighbours of a node $i$ the union of the neighbours of node $i$ in the various layers. Then the algorithm starts a loop, where each node is considered in order (for this reason the indexing of the nodes changes the output of the algorithm). Call $i$ the node taken into consideration and $C_i$ its community. The algorithm removes node $i$ from its community $C_i$ and calculates the modularity gain $\Delta Q_i$ on each layer.

For each neighbour $j$ of node $i$ (if the community of node $j$ has not already been considered), the method includes $i$ in the community of node $j$, called $C_j$, and calculates the corresponding modularity gain $\Delta Q_{i \rightarrow j}$ on each layer. Now the algorithm calculates the gain $\Delta F_{i \rightarrow j}$ of the defined function $F$, for changing the community of node $i$. Among all the positive gains of function $F$, which therefore give an increase of the function, the algorithm selects the highest one $\Delta F_{i \rightarrow j}$ and put node $i$ into the corresponding community $C_j$. The algorithm recalculates the modularity and the function values corresponding to this new partition. This first phase stops when it finds a local maxima of the function, i.e. when no individual move can improve the function value.

The second phase remains unchanged respect to the original method for single-layer graphs. The algorithm constructs a reduced network, where each community becomes a node, such that all-singleton partition has the same value of modularity as the partition that we identified at the end of 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, and links between nodes of the same community lead to self-loops for the community.

The algorithm then iterates the whole procedure, until the heuristic converges, i.e. until phase 2 induces no further changes. Look at Algorithm 1 for a pseudocode of the Louvain Expansion method.

Note that the output of the algorithm depends on the order of the nodes. Therefore, the way in which we index the new communities at the end of phase 2 is important. It seems that the ordering of the nodes does not influence significantly the final modularity, however it appears to affect the computational time. In the literature, this aspect is still unclear. We decide to order the nodes due to the community size.

The Louvain heuristic is very popular for its simplicity and efficiency. Part of the algorithm’s efficiency results from the fact that the modularity can be calculated iteratively during the procedure. At the beginning of phase 1, the method calculates the modularity from scratch. In Chapter 1 we proposed a formula to calculate modularity of an unweighted graph (equation (1.2)), however during phase 2 the algorithm creates a reduced graph with weighted edges,
Algorithm 1: Louvain Expansion

Input: $G$ multi-layer graph

Output: $C$ final assignment of nodes to communities

1: function LOUVAIN EXPANSION($G$)
2: repeat
3:   PHASE 1
4:   Initialize:
5:     $C \leftarrow$ initial partition, where each vertex of graph $G$ is a community
6:     $Q \leftarrow$ modularity vector of the initial partition
7:     $F \leftarrow$ function value of the initial partition
8:     $NB \leftarrow$ neighbour nodes vector
9:   repeat
10:     for each node $i$ do
11:       remove node $i$ from its community
12:       $\Delta Q_{i} \leftarrow$ modularity gain for removing node $i$ from its community
13:     end for
14:     for each node $j$ that is neighbour of node $i$ do
15:       insert node $i$ into community of node $j$
16:       $\Delta Q_{2i \rightarrow j} \leftarrow$ modularity gain for inserting $i$ into community of $j$
17:       $\Delta F_{i \rightarrow j} \leftarrow$ function gain for changing the community of node $i$
18:     end for
19:     $\Delta F_{i \rightarrow j*} \leftarrow$ best function gain
20:     if $\Delta F_{i \rightarrow j*} > 0$ then
21:       move node $i$ into community of node $j*$
22:     end if
23:   until no improved clustering found.
24:   PHASE 2
25:     $G \leftarrow$ reduced graph where each community of partition $C$ is a node
26: until no improved clustering found.
27: return $C$
28: end function

23
therefore we show also the corresponding formula for weighted graphs

\[ Q_w = \frac{1}{2w} \sum_{i,j} \left( W_{ij} - \frac{s_i s_j}{2w} \right) \delta(C_i, C_j) \quad (3.1) \]

where the sum runs over all pairs of vertices, \( W \) is the incidence matrix, \( w \) the sum of the weights of all the edges of the graph, \( s_i \) is the strength of node \( i \) (the sum of the weights of all the edges incident to \( i \)), and \( C_i \) is the community of node \( i \). The function \( \delta \) yields one if vertices \( i \) and \( j \) are in the same community \((C_i = C_j)\), zero otherwise.

From now we suppose to work with a unweighted graph, however each formula can be easily extended to the weighted case.

To calculate modularity, the algorithm actually uses this formula, that is equivalent to the equation (1.2),

\[ Q = \sum_{c \in C} \frac{|E(c)|}{m} - \left( \frac{\sum_{i \in N} k_i}{2m} \right)^2 \quad (3.2) \]

where \( C = (C_1, \ldots, C_n) \) is the clustering, \( m = |E| \) is the number of edges, \( |E(c)| \) is the sum of all the links between nodes in \( C \) and \( k_i \) is the degree of node \( i \).

During the loop, the algorithm calculates the gain in modularity in an easy way. The gain \( \Delta Q_{1i} \) obtained by moving a node \( i \) from its community \( c \) can easily be computed by

\[ \Delta Q_{1i} = \sum_{\text{tot}} k_i \frac{k_i}{2m^2} - \sum_{\text{tot}} \frac{k_i^2}{2m^2} - \frac{k_{i,\text{in}}}{m} \quad (3.3) \]

where \( \sum_{\text{tot}} \) is the sum of weights of the links incident to node in \( c \), \( k_i \) is the degree of node \( i \), \( k_{i,\text{in}} \) is the sum of weights of the links from \( i \) to nodes in \( c \), \( m \) is the sum of weights of all the links inside the network.

The following expression is used in order to evaluate the change of modularity \( \Delta Q_{2j} \) when an isolated node \( i \) is moved into the community \( c \) of node \( j \)

\[ \Delta Q_{2i \rightarrow j} = \sum_{i,\text{in}} \frac{k_i}{m} - \sum_{\text{tot}} \frac{k_i}{2m^2} \quad (3.4) \]

where \( \sum_{i,\text{in}} \) is the sum of weights of the links inside community \( c \), \( \sum_{\text{tot}} \) is the sum of weights of the links incident to node in \( c \), \( k_i \) is the degree of node \( i \), \( m \) is the sum of weights of all the links inside the network.

Thus, if a node \( i \) has changed community, the algorithm calculates the modularity of the new partition just adding to the initial modularity value the gains obtained above, rather than calculate it from scratch. Thanks to this fact, the Louvain heuristic is extremely fast.

In our method, it’s reasonable to use a function \( F \) that can include the information of the multiple layers and that can be calculated iteratively.

The most intuitive idea to extend the Louvain heuristic from the single-layer to the multi-layer case, already studied in the literature, is to take as function \( F \) the average of modularity on the layers

\[ M_Q = \frac{\sum_{s=1}^{k} Q_s}{k} \quad (3.5) \]
where $k$ is the number of layers and $Q_s$ is the modularity of layer $s$. The gain of this function can be calculated easily as follows:

$$\Delta M_Q = \frac{\sum_{s=1}^{k} \Delta Q_s}{k}$$

(3.6)

where $k$ is the number of layers and $\Delta Q_s$ is the gain on modularity of layer $s$.

We refer to this method with community-average (ComA). Matlab code of this method is available in Appendix A.

In real networks, different situations arise. We study two cases: the informative case, where each single layer has a piece of meaningful information from its own perspective, and the noisy case, where there are some noisy layers, which give us wrong information about communities. In order to analyse better these two situations, we propose two functions that take into account sample variance of modularity on the layers:

$$V_Q = \frac{\sum_{s=1}^{k} (Q_s - M_Q)^2}{k-1}$$

(3.7)

where $k$ is the number of layers, $Q_s$ is the modularity on layer $s$ and $M_Q$ is the average of modularity on the layers.

We suggest to take a convex combination of the average and the variance of modularity on the layers. In particular, we study these two functions:

$$F_- = (1 - \gamma) M_Q - \gamma V_Q$$

(3.8)

$$F_+ = (1 - \gamma) M_Q + \gamma V_Q$$

(3.9)

where $M_Q$ is the average of modularity on the layers, $V_Q$ is the variance of modularity on the layers and $\gamma \in [0, 1]$.

The idea behind is that: for the informative case we would like to maximize the average and minimize the variance of modularity on the layers, instead in the noisy case we want to maximize both the values.

The gain of both the functions can be easily calculated during the algorithm respectively in these ways:

$$\Delta F_- = (1 - \gamma) \Delta M_Q - \gamma \left( V_{\Delta Q} + \frac{2}{k-1} (Q - M_Q)^t (\Delta Q - \Delta M_Q) \right)$$

(3.10)

$$\Delta F_+ = (1 - \gamma) \Delta M_Q + \gamma \left( V_{\Delta Q} + \frac{2}{k-1} (Q - M_Q)^t (\Delta Q - \Delta M_Q) \right)$$

(3.11)

where $k$ is the number of layers, $Q$ is a vector in which the entrance $s$ is the initial modularity of layer $s$, $M_Q$ is the initial average of modularity on the layers, $\Delta Q$ is a vector in which the entrance $s$ is the gain in modularity of layer $s$, $\Delta M_Q$ is the gain of the average of modularity on the layers (that coincides with $M_{\Delta Q}$ the average of $\Delta Q$, as shown in equation (3.6)), $V_{\Delta Q}$ is the variance of $\Delta Q$, calculated as follows:

$$V_{\Delta Q} = \frac{\sum_{s=1}^{k} (\Delta Q_s - \Delta M_Q)^2}{k-1}$$

(3.12)
We refer to the method that uses the function $F_-$ as community-variance-minus (ComV-) and to the method that uses the function $F_+$ as community-variance-plus (ComV+). Matlab codes of these methods are available in Appendix B and Appendix C.

All the methods that we have introduced coincide with the original Louvain algorithm if applied to single-layer graphs.

### 3.2 Louvain Multiobjective

In this Section, we present the Louvain Multiobjective method for community detection in multi-layer networks, that is more sophisticated respect to the one described in Section 3.1. The algorithm is a filter type method that takes into account the multiobjective nature of the problem. It is based on the Louvain method for single-layer graphs [9] and tries to expand it to the multi-layer case. The basic idea is to not decide just one community to put in node $i$ during phase 1, but to follow more case studies.

The problem of maximizing modularity over multiple layers is a problem of multiobjective optimization. We consider a vector $Q=(Q_s)_{s=1,...,k}$ where $k$ is the number of layers, and we want to maximize simultaneously all its entries. If there are no conflicts between the entries, the straightforward optimal solution of the problem is obtained solving separately $k$ optimization problems. However, this is not what usually happens in real networks. In multiobjective optimization there is not a unique way to define the concept of optimality, since there is not a total order for $\mathbb{R}^k$. Each partial order defines a different definition of optimality. We adopt a definition that was proposed for the first time by Edgeworth in 1881 and later revised by Vilfredo Pareto in 1896 [91]. All the definitions are referred to a maximisation problem.

**Definition 1.** Given two vectors $z^1$ and $z^2 \in \mathbb{R}^k$, $z^1$ dominates $z^2$ according to Pareto, and we write $z^1 \geq^p z^2$, if

$$
z^1_i \geq z^2_i \quad \text{for each index } i=1,...,k \quad \text{and} \\
z^1_j > z^2_j \quad \text{for at least one index } j=1,...,k.
$$

This binary relation induces a partial order over $\mathbb{R}^k$. Thus we can give the definition of optimality according to Pareto.

**Definition 2.** A vector $z^* \in \mathbb{R}^k$ is Pareto optimal if there is not other vectors $z \in \mathbb{R}^k$ such that $z^* \leq^p z$.

The Pareto front is the set of all Pareto optimals.

We define a filter as a list of vectors such that no vector dominates the others.

The input of the Louvain Multiobjective algorithm is a multi-layer graph $G$ with $k$ layers. We suppose to have multi-layer graphs where just edges vary between layers, thus each node is present in all layers, and there are not edges between nodes of different layers. Each layer is an
undirected graph and just a single edge between nodes is allowed. The output of the algorithm is a final assignment of nodes to communities.

The algorithm is composed of two phases that are repeated iteratively. At the beginning of the first phase, each node forms a community. The algorithm calculates some values related to this first partition, s.a. $Q_s$ the modularity of the clustering in layer $s$ for $s=1,\ldots,k$ and a defined function $F$ connected to them. The method inserts the initial partition and the corresponding modularity vector in a filter $L$. We consider as neighbours of a node $i$ the union of the neighbours of node $i$ in the various layers. Then the algorithm starts a loop, where each node is considered in order (for this reason the indexing of the nodes changes the output of the algorithm). Call $i$ the node taken into consideration. For each partition in filter $L$, the method does the following procedure. Call $C_i$ the community of node $i$ in the considered partition. The algorithm removes node $i$ from its community $C_i$ and calculates the modularity gain $\Delta Q_i$ on each layer. For each neighbour $j$ of node $i$ (if the community of node $j$ has not already been considered), the method includes $i$ in the community of node $j$, called $C_j$, and calculates the corresponding modularity gain $\Delta Q_{i\rightarrow j}$ on each layer. Now the algorithm calculates the gain $\Delta F_{i\rightarrow j}$ of the defined function $F$, for changing the community of node $i$. If the function gain is positive, which therefore gives an increase of the function, the algorithm memorises the partition and the corresponding modularity vector in the filter. More precisely, the new modularity vector is added to $L$ only if it is not dominated and, if this condition is verified, the method deletes from the list all the modularity vectors that are now dominated by the new one. When the method has finished doing this procedure on each element of the initial filter, it checks the length of the new list $L$. If the filter is too long compared to a previously decided length $h$, the filter is cut removing the partitions with the least function values, until it is of the required length. This first phase stops when no moves change the filter. At this point, the method selects the partition of the filter with the maximum value of the function.

The second phase remains unchanged respect to the original method for single-layer graphs. The algorithm constructs a reduced network, where each community becomes a node, such that all-singleton partition has the same value of modularity as the partition that we identified at the end of 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, and links between nodes of the same community lead to self-loops for the community. The algorithm then iterates the whole procedure, until the heuristic converges, i.e. until phase 2 induces no further changes.

Look at Algorithm 2 for a pseudocode of the Louvain Multiobjective method.

Note that, also in this method, the output depends on the order of the nodes. We decide to order the nodes due to the community size to study the informative case, instead to index them following the initial order to study the noisy case, because this case has a higher computational time.

The method uses a filter because we want to get closer to the Pareto front. However, considering all the case studies would be too expensive in terms of computational time. For this reason, the method cuts the list to a length $h$ using as a criterion a function $F$. 

27
Algorithm 2 Louvain Multiobjective

Input: $G$ multi-layer graph
Output: $C$ final assignment of nodes to communities

function LOUVAIN_MULTIOBJECTIVE($G$)

repeat

PHASE 1

Initialize:
$C \leftarrow$ initial partition, where each vertex of graph $G$ is a community
$Q \leftarrow$ modularity vector of the initial partition
$F \leftarrow$ function value of the initial partition
$NB \leftarrow$ neighbour nodes vector
$L \leftarrow (C, Q)$ filter

repeat

for each node $i$ do
$L_{old} \leftarrow L$

for each element $(C, Q)$ in $L_{old}$ do
remove node $i$ from its community
$\Delta Q_i \leftarrow$ modularity gain for removing node $i$ from its community

for each node $j$ that is neighbour of node $i$ do
insert node $i$ into community of node $j$
$\Delta Q_{j,i} \leftarrow$ modularity gain for inserting $i$ into community of $j$
$\Delta F_{j,i} \leftarrow$ function gain for changing the community of node $i$

if $\Delta F_{j,i} > 0$ then
move node $i$ into community of node $j$: $C_{i \rightarrow j}$ new partition
$Q_{i \rightarrow j} \leftarrow$ modularity vector of the new partition

if $Q_{i \rightarrow j}$ not dominated by any element of the filter $L$ then
$L \leftarrow L \cup (C_{i \rightarrow j}, Q_{i \rightarrow j})$
$L \leftarrow L \setminus$ (elements dominated by $(C_{i \rightarrow j}, Q_{i \rightarrow j})$)
end if
end if
end for
end for

if length($L$) > $h$ then
remove from $L$ the partitions with the least function values, until it is of length $h$
end if
end repeat

until no changes of filter $L$

$(C^*, Q^*) \leftarrow$ element of the filter with the maximum value of the function

PHASE 2

$G \leftarrow$ reduced graph where each community of partition $C^*$ is a node

until no improved clustering found.

return $C$
end function
We study three variants of this method, taking as function $F$: the average of modularity on layers $M_Q$ described in (3.5), and the functions $F_-$ and $F_+$ defined in (3.8) and (3.9). We refer to these algorithms respectively as multi-average (MultiA), multi-variance-minus (MultiV-), multi-variance-plus (MultiV+). Matlab codes of these methods are available respectively in Appendix D, Appendix E and Appendix F.

As we have seen in Section 3.1, the algorithm calculates the modularity and the function values in a iterative and easy way.

Note that this method coincides with the method described in Section 3.1 when the filter has a unit length. Therefore, it is equal to the original Louvain algorithm if applied to single-layer graphs.
Chapter 4

Experimentation

We implemented the methods described in Chapter 3 using Matlab, starting from a available Matlab code for the Louvain heuristic for single layer graphs [92]. The Matlab codes are reported in the Appendix. We tested the algorithms using a machine equipped with i7 processor with 1.8 GHz and 16 GB of ram memory. The tests were performed both on artificial networks and on real world networks.

In this Chapter, we show the obtained results. Section 4.2 presents the results achieved on artificial graphs, and Section 4.3 reports the algorithm outputs obtained on real datasets. Preliminarily, in Section 4.1 we focus on how evaluate the partitions obtained as results of the methods, to later compare them. In particular, we used the accuracy and the Normalized Mutual Information.

4.1 Evaluation

In order to compare the results of the algorithms described in Chapter 3, we need to evaluate the partitions obtained as output. In the literature, various ways have been proposed on how to measure the performance of an algorithm for community detection. However, this question has not been fully resolved yet. We worked under the main assumption of knowing the community structure of the graph, that we call standard partition. On the other hand, we refer to the partitions obtained by the algorithms as predicted partitions. We evaluated the results of our algorithms through two values: the accuracy (AC) and the Normalized Mutual Information (NMI). To calculate both these values, we use indirectly the confusion matrix.

The confusion matrix $T$ is a table that allows visualization of the performance of an algorithm. It has a row for each community of the standard partition and a column for each community of the predicted partition. The element $T_{ij}$ represents the number of nodes in the $j$ community of the predicted partition that are in the $i$ community of the standard partition. The sum of the $i$ row (resp. column) gives us the total number of nodes of the $i$ community of the standard (resp. predicted) partition. The values on the diagonal are the nodes that are in the same community in both the partitions, instead the values off-diagonal are the nodes placed in the wrong community.
One disadvantage of the confusion matrix is that it depends on the labels of the communities.

The first way adopted to measure the results of the algorithms is the accuracy (Ac), i.e. the percentage of nodes placed in the right community. First of all, we calculated the number of nodes that are placed in the right community, using the confusion matrix and labelling the communities appropriately. We then considered the percentage of the nodes placed in the right community respect to the total number of nodes, in order to possibly compare results referred to graphs of different size.

We evaluated the output partitions obtained by the algorithms also through the Normalized Mutual Information (NMI), because we supposed that the community structures were known. Mutual Information is widely used in physics, statistics, and machine learning as a tool for comparing different labellings of a set of objects. In network science it is perhaps the standard measure for quantifying the performance of community detection algorithms. However, it can give inaccurate answers under certain conditions, s.a. when communities have different size. For this reason, Danon et al. proposed a normalization of this measure, called Normalized Mutual Information [93]. Given the standard partitioning \( C^* \) and the obtained partitioning \( C \), the Normalized Mutual Information can be calculated by the following formula

\[
\text{NMI}(C^*, C) = \frac{2 \cdot I(C^*, C)}{H(C^*) + H(C)}
\]

where \( H(C) \) is the entropy of partition \( C \) and \( I(C^*, C) \) is the Mutual Information between \( C^* \) and \( C \). This is defined by

\[
I(C^*, C) = H(C^*) - H(C^*|C)
\]

where \( H(C^*|C) \) is the conditional entropy of \( C^* \) respect to \( C \).

We calculated the NMI using the confusion matrix \( T \), through the formula

\[
\text{NMI}(C^*, C) = \frac{-2 \cdot \sum_{i=1}^{|C|} \sum_{j=1}^{C^*} T_{ij} \cdot \log\left(\frac{T_{ij}}{T_i \cdot T_j}t\right)}{\sum_{i=1}^{|C|} T_{i.} \cdot \log\left(\frac{T_i}{t}\right) + \sum_{j=1}^{|C^*|} T_{.j} \cdot \log\left(\frac{T_j}{t}\right)}
\]

where \( t = \text{sum}(T) \) is the number of nodes, \( T_{i.} \) is the sum of row \( i \) of \( T \) and \( T_{.j} \) is the sum of column \( j \) of \( T \).

NMI\((C^*, C)\) is a number between 0 and 1. It is equal to 1 when the two partitions are identical and equal to 0 when the two partitions are totally independent, s.a. when \( C \) is the single community that includes the whole graph.

A nice feature of NMI is that it is invariant under permutations of the labels of the communities.

### 4.2 Artificial Networks

We tested the algorithms described in Chapter 3 on artificial networks. In this Section, we show and compare the obtained results.
The Stochastic Block Model (SBM) is a generative model for graphs showing certain clusters structures through the parameters $p_{\text{in}}$ and $p_{\text{out}}$. These parameters represent the edge probabilities: given nodes $v_i$ and $v_j$ the probability of observing an edge between them is $p_{\text{in}}$ (resp. $p_{\text{out}}$), if $v_i$ and $v_j$ belong to the same (resp. different) cluster.

We analysed two different settings: in the first one all layers have the same class structure, in the second setting one layer is informative and the remaining layers are just noise. We set $p_{\text{in}} >> p_{\text{out}}$ on the informative layers and $p_{\text{in}} = p_{\text{out}}$ on the noisy layers.

In particular, we created networks with 4 communities of 125 nodes each and with $k = 2, 3$ layers, by fixing $p_{\text{in}} = 0.1$ and varying $p_{\text{out}}$. In the noisy layers, we fixed $p_{\text{in}} = p_{\text{out}} = 0.1$.

For each case, we report the results in a table. Each row corresponds to a method, which is indicated in the first column. We studied the Louvain Multiobjective models for length of the filter $h = 2, 3$ and, in the definition of function $F_i$ in equation (3.8) and $F_{i+}$ in equation (3.9), we set $\gamma = 0.1, 0.3, 0.5$. The ratio between $p_{\text{in}}$ and $p_{\text{out}}$ changes between columns and it is specified in the second row. For each method and ratio between $p_{\text{in}}$ and $p_{\text{out}}$, the tables report the accuracy (Ac) in percentage, the Normalized Mutual Information (NMI) and the execution time in seconds (Cpu) of the corresponding output. The best performances are marked with bold fonts and gray background and second best performances with only gray background. We show the results also using bar plots. In the tables and in the bar plots, we report the average of the values on 10 runs. We summarize the results of the multiple runs through some boxplots.

### 4.2.1 Informative case

In the informative case all layers have the same community structure, so each single layer has a piece of meaningful information.

For this case, we compared the models community-average (ComA) and community-variance-minus (ComV-) (Section 3.1), multi-average (MultiA) and multi-variance-minus (MultiV-) (Section 3.2). The idea behind is that for the informative case we would like to maximize the average and minimize the variance of modularity on the layers.

Table 4.1 shows the average results for the informative case on graphs with $k = 2$ layers. Figure 4.1 and Figure 4.2 represent the Accuracy and the NMI of the results in some bar plots. We summarize the results of the 10 runs in the boxplots in Figure 4.3 and Figure 4.4.

$p_{\text{in}}$ is fixed equal to 0.1 and $p_{\text{out}}$ varies according to the ratio between $p_{\text{in}}$ and $p_{\text{out}}$ equal to 3, 2.5, 2, 1.5.

The results get worse as the ratio between $p_{\text{in}}$ and $p_{\text{out}}$ decreases, in fact intuitively the communities are less defined when $p_{\text{out}}$ is close to $p_{\text{in}}$. For this reason, the execution times behave the same way, increasing as the ratio decreases. In particular, the Louvain Multiobjective methods are slower than the others.

All the methods perform very good in the first two cases with $p_{\text{in}}/p_{\text{out}} = 3, 2.5$, a little less well in the third case with $p_{\text{in}}/p_{\text{out}} = 2$ and get worst in the last situation with $p_{\text{in}}/p_{\text{out}} = 1.5$. 

33
The algorithms show similar results unless for $p_{in}/p_{out} = 2$, where the better performances are obtained by MultiV- with $h = 2$, $\gamma = 0.3$ and 0.5.

We studied in depth the more critical interval $[2.5, 2]$, sampling it. Table 4.2 shows the performances to vary of $p_{in}/p_{out} = 2.6, 2.4, 2.2, 2$. The methods still perform approximately the same way. Figure 4.5 and Figure 4.6 represent the Accuracy and the NMI of the results in some bar plots. We summarize the results of the 10 runs in the boxplots in Figure 4.7 and Figure 4.8.

Table 4.3 shows the average results for the informative case on graphs with $k = 3$ layers. Figure 4.9 and Figure 4.10 represent the Accuracy and the NMI of the results in some bar plots. We summarize the results of the 10 runs in the boxplots in Figure 4.11 and Figure 4.12.

$p_{in}$ is fixed equal to 0.1 and $p_{out}$ varies according to the ratio between $p_{in}$ and $p_{out}$ equal to 3, 2.5, 2, 1.5.

Also in this case, the results get worse as the ratio between $p_{in}$ and $p_{out}$ decreases, and the execution times behave the same way. In particular, the Louvain Multiobjective methods are slower than the others.

All the methods perform very good and similar for values of $p_{in}/p_{out} = 3, 2.5, 2$, but unfortunately all of them get worst in the last case with $p_{in}/p_{out} = 1.5$.

We analysed better the more interesting interval $[2, 1.5]$ in Table 4.4, taking $p_{in}/p_{out} = 2, 1.9, 1.8, 1.7$. The algorithms obtain almost the same results for the first two values; in the third case MultiV- with $h = 2$, $\gamma = 0.1$ and ComA achieve the best results; finally in the last case MultiA with $h = 3$ definitely outperforms all the other methods. Figure 4.13 and Figure 4.14 represent the Accuracy and the NMI of the results in some bar plots. We summarize the results of the 10 runs in the boxplots in Figure 4.15 and Figure 4.16.

Compared to the case with 2 layers, all the methods give best results. In fact, in the informative case, have multiple layers correspond to have more information. Nevertheless, execution times are higher for graphs with 3 layers.

In general, in the informative case on graphs with both 2 and 3 layers: when the community structure is well defined, all the algorithms perform well; when it is more confused, they give different outputs, but there is not a method that always dominated all the others.

### 4.2.2 Noisy case

In the noisy case just one layer is informative, so it has a community structure, and all the other layers are just noise, so they give wrong information about the clustering.

For this case, we compared the models community-average (ComA) and community-variance-plus (ComV+) (Section 3.1), multi-average (MultiA) and multi-variance-plus (MultiV+) (Section 3.2). The idea behind is that for the noisy case we would like to maximize both the average and the variance of modularity on the layers.

Table 4.5 shows the average results for the noisy case on graphs with $k = 2$ layers. Figure 4.17 and Figure 4.18 represent the Accuracy and the NMI of the results in some bar plots. We
### INFORMATIVE CASE: k=2

<table>
<thead>
<tr>
<th></th>
<th>$p_{in}/p_{out}=3$</th>
<th>$p_{in}/p_{out}=2.5$</th>
<th>$p_{in}/p_{out}=2$</th>
<th>$p_{in}/p_{out}=1.5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
<td>Ac</td>
</tr>
<tr>
<td>ComA</td>
<td>99.74</td>
<td>0.989</td>
<td>2.52</td>
<td>99.04</td>
</tr>
<tr>
<td>MultiA</td>
<td>99.76</td>
<td>0.990</td>
<td>140.72</td>
<td>98.84</td>
</tr>
<tr>
<td>MultiA</td>
<td><strong>99.80</strong></td>
<td><strong>0.992</strong></td>
<td>147.78</td>
<td>98.92</td>
</tr>
<tr>
<td>MultiA</td>
<td>99.76</td>
<td>0.990</td>
<td>2.86</td>
<td>98.84</td>
</tr>
<tr>
<td>MultiA</td>
<td>99.76</td>
<td>0.990</td>
<td>3.46</td>
<td>98.76</td>
</tr>
<tr>
<td>MultiA</td>
<td>99.74</td>
<td>0.990</td>
<td>6.05</td>
<td>98.74</td>
</tr>
<tr>
<td>MultiV-$\gamma=0.5$</td>
<td>99.76</td>
<td>0.990</td>
<td>163.26</td>
<td>98.80</td>
</tr>
<tr>
<td>MultiV-$\gamma=0.1, h=2$</td>
<td><strong>99.80</strong></td>
<td><strong>0.992</strong></td>
<td>170.17</td>
<td>98.96</td>
</tr>
<tr>
<td>MultiV-$\gamma=0.1, h=3$</td>
<td>99.80</td>
<td>0.992</td>
<td>169.90</td>
<td>98.98</td>
</tr>
<tr>
<td>MultiV-$\gamma=0.3, h=2$</td>
<td>99.72</td>
<td>0.988</td>
<td>197.79</td>
<td>98.96</td>
</tr>
<tr>
<td>MultiV-$\gamma=0.3, h=3$</td>
<td>99.78</td>
<td>0.991</td>
<td>106.79</td>
<td>98.74</td>
</tr>
<tr>
<td>MultiV-$\gamma=0.5, h=2$</td>
<td>99.78</td>
<td>0.991</td>
<td>106.79</td>
<td>98.74</td>
</tr>
<tr>
<td>MultiV-$\gamma=0.5, h=3$</td>
<td>99.76</td>
<td>0.990</td>
<td>185.08</td>
<td>98.92</td>
</tr>
</tbody>
</table>

Table 4.1: Experiments in the informative case on artificial graphs with k=2 layers. Notation: **best** performances are marked with bold fonts and gray background and **second** best performances with only gray background.
Figure 4.1: Accuracy of the experiments in the informative case on artificial graphs with k=2 layers
Figure 4.2: NMI of the experiments in the informative case on artificial graphs with k=2 layers.
Figure 4.3: Accuracy of the experiments in the informative case on artificial graphs with k=2 layers
Figure 4.4: NMI of the experiments in the informative case on artificial graphs with k=2 layers
### INFORMATIVE CASE: $k=2$

<table>
<thead>
<tr>
<th></th>
<th>$p_{in}/p_{out}=2.6$</th>
<th></th>
<th>$p_{in}/p_{out}=2.4$</th>
<th></th>
<th>$p_{in}/p_{out}=2.2$</th>
<th></th>
<th>$p_{in}/p_{out}=2$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
<td>Ac</td>
<td>NMI</td>
</tr>
<tr>
<td>ComA</td>
<td>98.90</td>
<td>0.956</td>
<td>3.13</td>
<td>98.32</td>
<td>0.936</td>
<td>3.99</td>
<td>93.28</td>
<td>0.833</td>
</tr>
<tr>
<td>MultiA h=2</td>
<td>98.94</td>
<td><strong>0.975</strong></td>
<td>139.63</td>
<td>98.30</td>
<td>0.934</td>
<td>197.19</td>
<td>96.78</td>
<td>0.888</td>
</tr>
<tr>
<td>MultiA h=3</td>
<td><strong>98.96</strong></td>
<td>0.958</td>
<td>227.07</td>
<td>98.42</td>
<td>0.939</td>
<td>276.52</td>
<td>97.16</td>
<td>0.895</td>
</tr>
<tr>
<td>ComV- $\gamma=0.1$</td>
<td>98.90</td>
<td>0.956</td>
<td>4.20</td>
<td>98.32</td>
<td>0.935</td>
<td>4.90</td>
<td><strong>97.26</strong></td>
<td><strong>0.898</strong></td>
</tr>
<tr>
<td>ComV- $\gamma=0.3$</td>
<td>98.92</td>
<td>0.956</td>
<td>4.29</td>
<td>98.26</td>
<td>0.933</td>
<td>4.92</td>
<td>97.00</td>
<td>0.891</td>
</tr>
<tr>
<td>ComV- $\gamma=0.5$</td>
<td>98.94</td>
<td>0.957</td>
<td>4.59</td>
<td><strong>98.44</strong></td>
<td><strong>0.940</strong></td>
<td>4.65</td>
<td><strong>97.20</strong></td>
<td><strong>0.896</strong></td>
</tr>
<tr>
<td>MultiV- $\gamma=0.1$, h=2</td>
<td>98.92</td>
<td>0.956</td>
<td>144.55</td>
<td><strong>98.46</strong></td>
<td><strong>0.941</strong></td>
<td>200.38</td>
<td>96.70</td>
<td>0.880</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.1$, h=3</td>
<td><strong>98.96</strong></td>
<td>0.958</td>
<td>216.71</td>
<td>98.36</td>
<td>0.937</td>
<td>286.79</td>
<td>97.06</td>
<td>0.890</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.3$, h=2</td>
<td><strong>99.00</strong></td>
<td><strong>0.959</strong></td>
<td>180.89</td>
<td>98.30</td>
<td>0.935</td>
<td>192.75</td>
<td>96.52</td>
<td>0.885</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.3$, h=3</td>
<td>98.94</td>
<td>0.957</td>
<td>240.51</td>
<td>98.32</td>
<td>0.936</td>
<td>287.64</td>
<td>95.86</td>
<td>0.874</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.5$, h=2</td>
<td><strong>99.00</strong></td>
<td><strong>0.959</strong></td>
<td>144.67</td>
<td>98.42</td>
<td>0.939</td>
<td>188.69</td>
<td>96.98</td>
<td>0.888</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.5$, h=3</td>
<td><strong>99.00</strong></td>
<td><strong>0.959</strong></td>
<td>231.16</td>
<td>98.20</td>
<td>0.931</td>
<td>280.32</td>
<td>96.82</td>
<td>0.886</td>
</tr>
</tbody>
</table>

Table 4.2: Experiments in the informative case on artificial graphs with $k=2$ layers. Notation: **best** performances are marked with bold fonts and gray background and **second** best performances with only gray background.
Figure 4.5: Accuracy of the experiments in the informative case on artificial graphs with k=2 layers

(a) $p_{in}/p_{out}=2.6$

(b) $p_{in}/p_{out}=2.4$

(c) $p_{in}/p_{out}=2.2$

(d) $p_{in}/p_{out}=2$
Figure 4.6: NMI of the experiments in the informative case on artificial graphs with k=2 layers
Figure 4.7: Accuracy of the experiments in the informative case on artificial graphs with $k=2$ layers

(a) $p_{in}/p_{out}=2.6$

(b) $p_{in}/p_{out}=2.4$

(c) $p_{in}/p_{out}=2.2$

(d) $p_{in}/p_{out}=2$
Figure 4.8: NMI of the experiments in the informative case on artificial graphs with k=2 layers
<table>
<thead>
<tr>
<th></th>
<th>( \gamma = 0.1 )</th>
<th>( \gamma = 0.3 )</th>
<th>( \gamma = 0.5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Com</strong></td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
</tr>
<tr>
<td><strong>A</strong> h=2</td>
<td>99.99</td>
<td>0.996</td>
<td>5.11</td>
</tr>
<tr>
<td><strong>Multi</strong> h=2</td>
<td>99.99</td>
<td>0.996</td>
<td>231.03</td>
</tr>
<tr>
<td><strong>Multi</strong> h=3</td>
<td>99.99</td>
<td>0.996</td>
<td>358.93</td>
</tr>
<tr>
<td><strong>ComV</strong> ( \gamma = 0.1 )</td>
<td>99.99</td>
<td>0.996</td>
<td>5.94</td>
</tr>
<tr>
<td><strong>ComV</strong> ( \gamma = 0.3 )</td>
<td>99.99</td>
<td>0.996</td>
<td>6.09</td>
</tr>
<tr>
<td><strong>ComV</strong> ( \gamma = 0.5 )</td>
<td>99.99</td>
<td>0.996</td>
<td>5.67</td>
</tr>
<tr>
<td><strong>MultiV</strong> ( \gamma = 0.1, h=2 )</td>
<td>99.98</td>
<td>0.995</td>
<td>352.77</td>
</tr>
<tr>
<td><strong>MultiV</strong> ( \gamma = 0.1, h=3 )</td>
<td>99.99</td>
<td>0.996</td>
<td>273.54</td>
</tr>
<tr>
<td><strong>MultiV</strong> ( \gamma = 0.3, h=2 )</td>
<td>99.99</td>
<td>0.996</td>
<td>354.30</td>
</tr>
<tr>
<td><strong>MultiV</strong> ( \gamma = 0.5, h=2 )</td>
<td>99.99</td>
<td>0.997</td>
<td>234.06</td>
</tr>
<tr>
<td><strong>MultiV</strong> ( \gamma = 0.5, h=3 )</td>
<td>99.99</td>
<td>0.996</td>
<td>346.97</td>
</tr>
</tbody>
</table>

Table 4.3: Experiments in the informative case on artificial graphs with k=3 layers. Notation: **best** performances are marked with bold fonts and gray background and **second** best performances with only gray background.

INFORMATIVE CASE: k=3
Figure 4.9: Accuracy of the experiments in the informative case on artificial graphs with k=3 layers
Figure 4.10: NMI of the experiments in the informative case on artificial graphs with k=3 layers
Figure 4.11: Accuracy of the experiments in the informative case on artificial graphs with \( k=3 \) layers
Figure 4.12: NMI of the experiments in the informative case on artificial graphs with $k=3$ layers.
<table>
<thead>
<tr>
<th></th>
<th>$p_{in}/p_{out}=2$</th>
<th>$p_{in}/p_{out}=1.9$</th>
<th>$p_{in}/p_{out}=1.8$</th>
<th>$p_{in}/p_{out}=1.7$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
<td>Ac</td>
</tr>
<tr>
<td>ComA</td>
<td>98.80</td>
<td>0.952</td>
<td>5.42</td>
<td>98.02</td>
</tr>
<tr>
<td>MultiA</td>
<td>98.80</td>
<td>0.952</td>
<td>233.60</td>
<td>98.16</td>
</tr>
<tr>
<td>MultiA</td>
<td>98.80</td>
<td>0.952</td>
<td>357.67</td>
<td>98.10</td>
</tr>
<tr>
<td>ComV- $\gamma=0.1$</td>
<td>98.72</td>
<td>0.949</td>
<td>6.23</td>
<td>97.76</td>
</tr>
<tr>
<td>ComV- $\gamma=0.3$</td>
<td>98.74</td>
<td>0.950</td>
<td>6.36</td>
<td>97.90</td>
</tr>
<tr>
<td>ComV- $\gamma=0.5$</td>
<td>98.72</td>
<td>0.949</td>
<td>6.29</td>
<td>98.04</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.1, h=2$</td>
<td>98.70</td>
<td>0.948</td>
<td>238.61</td>
<td>98.02</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.1, h=3$</td>
<td><strong>98.84</strong></td>
<td>0.953</td>
<td>369.92</td>
<td><strong>98.18</strong></td>
</tr>
<tr>
<td>MultiV- $\gamma=0.3, h=2$</td>
<td>98.86</td>
<td>0.954</td>
<td>231.57</td>
<td><strong>98.18</strong></td>
</tr>
<tr>
<td>MultiV- $\gamma=0.3, h=3$</td>
<td>98.78</td>
<td>0.952</td>
<td>365.66</td>
<td>97.90</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.5, h=2$</td>
<td>98.80</td>
<td>0.952</td>
<td>241.91</td>
<td>98.10</td>
</tr>
<tr>
<td>MultiV- $\gamma=0.5, h=3$</td>
<td>98.82</td>
<td>0.953</td>
<td>356.77</td>
<td>97.96</td>
</tr>
</tbody>
</table>

Table 4.4: Experiments in the informative case on artificial graphs with k=3 layers. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.
Figure 4.13: Accuracy of the experiments in the informative case on artificial graphs with $k=3$ layers

(a) $p_{in}/p_{out}=2$

(b) $p_{in}/p_{out}=1.9$

(c) $p_{in}/p_{out}=1.8$

(d) $p_{in}/p_{out}=1.7$
Figure 4.14: NMI of the experiments in the informative case on artificial graphs with k=3 layers
Figure 4.15: Accuracy of the experiments in the informative case on artificial graphs with k=3 layers
Figure 4.16: NMI of the experiments in the informative case on artificial graphs with k=3 layers
summarize the results of the 10 runs in the boxplots in Figure 4.19 and Figure 4.20. 

\( p_{\text{in}} \) is fixed equal to 0.1 and \( p_{\text{out}} \) varies according to the ratio between \( p_{\text{in}} \) and \( p_{\text{out}} \) equal to 3, 2.5, 2.

The results get worse as the ratio between \( p_{\text{in}} \) and \( p_{\text{out}} \) decreases, in fact intuitively the communities are less defined when \( p_{\text{out}} \) is close to \( p_{\text{in}} \). For this reason, the execution times behave the same way, increasing as the ratio decreases. In particular, the Louvain Multiobjective methods are slower than the others.

ComV+ outperforms all the other methods in the first two cases with \( p_{\text{in}} / p_{\text{out}} = 3, 2.5 \). All the algorithm gives bad results in the last case with \( p_{\text{in}} / p_{\text{out}} = 2 \).

Table 4.6 shows the average results for the noisy case on graphs with \( k = 3 \) layers. Figure 4.21 and Figure 4.22 represent the Accuracy and the NMI of the results in some bar plots. We summarize the results of the 10 runs in the boxplots in Figure 4.23 and Figure 4.24.

\( p_{\text{in}} \) is fixed equal to 0.1 and \( p_{\text{out}} \) varies according to the ratio between \( p_{\text{in}} \) and \( p_{\text{out}} \) equal to 3, 2.5, 2, 1.5.

Also in this case, the results get worse as the ratio between \( p_{\text{in}} \) and \( p_{\text{out}} \) decreases, and the execution times behave the same way. In particular, the Louvain Multiobjective methods are slower than the others.

MultiV+ with \( \gamma = 0.5, h = 3 \) achieves the best result in the first situation with \( p_{\text{in}} / p_{\text{out}} = 3 \). All the algorithms obtained bad results in all the other cases. Compared to the case with 2 layers, all the methods give worst results. In fact, in the noisy case, just the first layer is informative and all the others are noise, so the noise increases as the number of layer increases. Moreover, execution times are higher for graphs with \( k = 3 \) layers.

In general, in the noisy case on graphs with both 2 and 3 layers, almost all the methods overcome ComA, and the best performances are obtained by the methods that take into consideration the variance of modularity on the layers as well as the average, giving them the same weight in equation (3.9).

All the algorithms achieve better results in the informative case then the noisy one. Moreover, execution times are lower in the informative case than in the noisy case. In fact, in the informative case each layer has a piece of meaningful information, instead in the noisy case some layers can give wrong information.

### 4.3 Real World Networks

In this Section, we compare the performances of the proposed approaches on real networks. We considered three real datasets: 3sources [94], BBCSports [95] and Wikipedia [96]. We used the corresponding layer matrices provided by Mercado et al. [97], which have used a similarity measure and have considered the unweighted version of the symmetric k-nearest neighbour graph (i.e. they have taken the k neighbours with highest correlation). Also in this case, we knew the community structure of the graphs.

3sources dataset corresponds to new articles of BBC, Reuters and Guardian. It produces a
Table 4.5: Experiments in the noisy case on artificial graphs with k=2 layers. Notation: best performances are marked with bold fonts and gray background and second best performances with only gray background.
Figure 4.17: Accuracy of the experiments in the noisy case on artificial graphs with $k=2$ layers

(a) $p_{in}/p_{out}=3$

(b) $p_{in}/p_{out}=2.5$

(c) $p_{in}/p_{out}=2$
Figure 4.18: NMI of the experiments in the noisy case on artificial graphs with k=2 layers
Figure 4.19: Accuracy of the experiments in the noisy case on artificial graphs with k=2 layers
Figure 4.20: NMI of the experiments in the noisy case on artificial graphs with $k=2$ layers
### NOISY CASE: k=3

<table>
<thead>
<tr>
<th></th>
<th>$p_{in}/p_{out}=3$</th>
<th>$p_{in}/p_{out}=2.5$</th>
<th>$p_{in}/p_{out}=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ComA</strong></td>
<td>Ac 31.08</td>
<td>NMI 0.140</td>
<td>Cpu 6.61</td>
</tr>
<tr>
<td><strong>MultiA</strong></td>
<td>h=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>35.40</td>
<td>0.183</td>
<td>366.97</td>
</tr>
<tr>
<td><strong>MultiA</strong></td>
<td>h=3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>37.36</td>
<td>0.196</td>
<td>8.67</td>
</tr>
<tr>
<td><strong>ComV+</strong></td>
<td>$\gamma=0.1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>38.68</td>
<td>0.219</td>
<td>8.58</td>
</tr>
<tr>
<td><strong>ComV+</strong></td>
<td>$\gamma=0.3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>52.64</strong></td>
<td><strong>0.328</strong></td>
<td>9.88</td>
</tr>
<tr>
<td><strong>MultiV+</strong></td>
<td>$\gamma=0.1$, h=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>36.62</td>
<td>0.188</td>
<td>245.82</td>
</tr>
<tr>
<td><strong>MultiV+</strong></td>
<td>$\gamma=0.1$, h=3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32.32</td>
<td>0.151</td>
<td>348.44</td>
</tr>
<tr>
<td><strong>MultiV+</strong></td>
<td>$\gamma=0.3$, h=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>42.10</td>
<td>0.227</td>
<td>225.27</td>
</tr>
<tr>
<td><strong>MultiV+</strong></td>
<td>$\gamma=0.3$, h=3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>36.00</td>
<td>0.189</td>
<td>373.14</td>
</tr>
<tr>
<td><strong>MultiV+</strong></td>
<td>$\gamma=0.5$, h=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>48.02</td>
<td>0.306</td>
<td>253.78</td>
</tr>
<tr>
<td><strong>MultiV+</strong></td>
<td>$\gamma=0.5$, h=3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>59.48</strong>, <strong>0.402</strong></td>
<td>362.97</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Experiments in the noisy case on artificial graphs with $k=3$ layers. Notation: **best** performances are marked with bold fonts and gray background and **second best** performances with only gray background.
Figure 4.21: Accuracy of the experiments in the noisy case on artificial graphs with k=3 layers
Figure 4.22: NMI of the experiments in the noisy case on artificial graphs with $k=3$ layers
Figure 4.23: Accuracy of the experiments in the noisy case on artificial graphs with k=3 layers
Figure 4.24: NMI of the experiments in the noisy case on artificial graphs with k=3 layers
multi-layers graph with 169 nodes, 3 layers, and 6 communities of size 56, 21, 11, 18, 51, 12. 

*BBCSports* is a dataset of sports articles of BBC and corresponds to a multi-layer graph with 544 nodes, 2 layers and 5 communities of size 62, 104, 193, 124, 61.

The last dataset, called *Wikipedia*, reports Wikipedia articles and gives a multi-layer graph with 693 nodes, 2 layers and 10 communities of size 34, 88, 96, 65, 58, 51, 41, 71, 104.

We analysed the informative and the noisy settings. In the noisy case, we kept the first layer and we added to all the other layers a matrix, generated by the Stochastic Block Model with $p_{in} = p_{out} = 0.05$. For each dataset, we tested the algorithms on 10 runs.

For each of the two cases, we report the results in a table. Each row corresponds to a method, which is indicated in the first column. We studied the Louvain Multiobjective models for length of the filter $h = 2, 3$ and, in the definition of function $F_-$ in equation (3.8) and $F_+$ in equation (3.9), we set $\gamma = 0.1, 0.3, 0.5$. The names of the datasets are reported in the second row. For each real dataset we show the accuracy (Ac) in percentage, the Normalized Mutual Information (NMI) and the execution time in seconds (Cpu) of the corresponding output. The best performances are marked with bold fonts and gray background and second best performances with only gray background. We show the results also using bar plots. In the noisy case, in the tables and in the bar plots, we report the average of the values on 10 runs. We summarize the results of the multiple runs through some boxplots.

### 4.3.1 Informative case

In the informative case, all layers have the same community structure, so each single layer has a piece of meaningful information.

For this case, we compared the models *community-average* (ComA) and *community-variance-minus* (ComV-) (Section 3.1), *multi-average* (MultiA) and *multi-variance-minus* (MultiV-) (Section 3.2). The idea behind is that for the informative case we would like to maximize the average and minimize the variance of modularity on the layers.

Table 4.7 shows the results for the informative case. Figure 4.25 and Figure 4.26 represent the Accuracy and the NMI of the results in some bar plots.

All the methods perform very good on the first two datasets 3sources and BBCSports. ComA performs the worst results. The best performances are achieved by MultiV- with $h = 2$, $\gamma = 0.5$ in the first dataset and by MultiV- with $h = 3$, $\gamma = 0.1$ in the second one. In the Wikipedia dataset the results are slightly worse and the highest outputs are obtained by MultiA with $h = 3$, ComV- with $\gamma = 0.3$ and ComA.

Execution times increase with increasing number of nodes and are higher for the Louvain Multiobjective methods.
4.3.2 Noisy case

In the noisy case, we kept the first layer and we added to all the other layers a matrix, generated by the Stochastic Block Model with $p_{in} = p_{out} = 0.05$. For each dataset, we tested the algorithms on 10 runs.

For this case, we compared the models community-average (ComA) and community-variance-plus (ComV+) (Section 3.1), multi-average (MultiA) and multi-variance-plus (MultiV+) (Section 3.2). The idea behind is that for the noisy case we would like to maximize both the average and the variance of modularity on the layers.

Table 4.8 shows the average results for the noisy case. Figure 4.27 and Figure 4.28 represent the Accuracy and the NMI of the results in some bar plots. We summarize the results of the 10 runs in the boxplots in Figure 4.29 and Figure 4.30.

All the methods perform well on the first two datasets 3sources and BBCSports. The worst results are obtained by algorithms that use function $F_+$ with $\gamma = 0.3$ and $0.5$. Unfortunately, all the methods give bad results for the Wikipedia dataset.

All the algorithms achieve better results in the informative case than the noisy one. Moreover, execution times are lower in the noisy case than in the informative case. In fact, in the informative case each layer has a piece of meaningful information, in fact in the noisy case some layers can give wrong information.

As seen in the informative case, execution times increase with increasing number of nodes and are higher for the Louvain Multiobjective methods.
<table>
<thead>
<tr>
<th></th>
<th>3sources</th>
<th></th>
<th>BBCSport</th>
<th></th>
<th>Wikipedia</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
</tr>
<tr>
<td>ComA</td>
<td>85.80</td>
<td>0.749</td>
<td>0.21</td>
<td>74.82</td>
<td>0.753</td>
<td>2.09</td>
</tr>
<tr>
<td>MultiA h=2</td>
<td>86.39</td>
<td>0.765</td>
<td>3.59</td>
<td>89.89</td>
<td>0.825</td>
<td>51.44</td>
</tr>
<tr>
<td>MultiA h=3</td>
<td>86.98</td>
<td>0.773</td>
<td>5.30</td>
<td>79.04</td>
<td>0.791</td>
<td>85.77 <strong>55.84</strong></td>
</tr>
<tr>
<td>ComV-γ=0.1</td>
<td>85.80</td>
<td>0.749</td>
<td>0.25</td>
<td>82.35</td>
<td>0.789</td>
<td>2.54</td>
</tr>
<tr>
<td>ComV-γ=0.3</td>
<td>85.80</td>
<td>0.749</td>
<td>0.26</td>
<td>75.18</td>
<td>0.751</td>
<td>2.40</td>
</tr>
<tr>
<td>ComV-γ=0.5</td>
<td>86.98</td>
<td>0.781</td>
<td>0.24</td>
<td>83.27</td>
<td>0.798</td>
<td>2.69</td>
</tr>
<tr>
<td>MultiV-γ=0.1, h=2</td>
<td>86.39</td>
<td>0.765</td>
<td>3.69</td>
<td>84.38</td>
<td>0.784</td>
<td>56.48</td>
</tr>
<tr>
<td>MultiV-γ=0.1, h=3</td>
<td>86.98</td>
<td>0.773</td>
<td>5.60</td>
<td><strong>90.44 0.837</strong></td>
<td>89.38</td>
<td>54.40</td>
</tr>
<tr>
<td>MultiV-γ=0.3, h=2</td>
<td>86.39</td>
<td>0.765</td>
<td>3.64</td>
<td>84.56</td>
<td>0.787</td>
<td>56.22</td>
</tr>
<tr>
<td>MultiV-γ=0.3, h=3</td>
<td><strong>88.76 0.812</strong></td>
<td>5.97</td>
<td>80.51</td>
<td>0.787</td>
<td>70.36</td>
<td>44.30</td>
</tr>
<tr>
<td>MultiV-γ=0.5, h=2</td>
<td><strong>88.76 0.812</strong></td>
<td>4.13</td>
<td>80.88</td>
<td>0.784</td>
<td>50.78</td>
<td>37.09</td>
</tr>
<tr>
<td>MultiV-γ=0.5, h=3</td>
<td>86.98</td>
<td>0.775</td>
<td>8.12</td>
<td>86.40</td>
<td>0.816</td>
<td>82.10</td>
</tr>
</tbody>
</table>

Table 4.7: Experiments in the informative case on real datasets. Notation: **best** performances are marked with bold fonts and gray background and **second** best performances with only gray background.
Figure 4.25: Accuracy of the experiments in the informative case on real datasets
Figure 4.26: NMI of the experiments in the informative case on real datasets
<table>
<thead>
<tr>
<th></th>
<th>3sources</th>
<th></th>
<th>BBCSport</th>
<th></th>
<th>Wikipedia</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
<td>Ac</td>
<td>NMI</td>
<td>Cpu</td>
</tr>
<tr>
<td>ComA</td>
<td>75.50</td>
<td>0.706</td>
<td>1.01</td>
<td>81.18</td>
<td>0.749</td>
<td>4.49</td>
</tr>
<tr>
<td>MultiA</td>
<td>76.80</td>
<td>0.715</td>
<td>14.48</td>
<td>81.84</td>
<td>0.752</td>
<td>128.82</td>
</tr>
<tr>
<td>MultiA</td>
<td>77.04</td>
<td><strong>0.717</strong></td>
<td><strong>22.32</strong></td>
<td><strong>83.25</strong></td>
<td><strong>0.757</strong></td>
<td><strong>192.67</strong></td>
</tr>
<tr>
<td>ComV+</td>
<td>77.63</td>
<td><strong>0.717</strong></td>
<td>1.20</td>
<td>80.74</td>
<td>0.739</td>
<td>5.29</td>
</tr>
<tr>
<td>ComV+</td>
<td>74.73</td>
<td>0.698</td>
<td>1.28</td>
<td>71.10</td>
<td>0.686</td>
<td>4.97</td>
</tr>
<tr>
<td>ComV+</td>
<td>69.53</td>
<td>0.661</td>
<td>1.09</td>
<td>65.09</td>
<td>0.648</td>
<td>5.03</td>
</tr>
<tr>
<td>MultiV+</td>
<td><strong>77.87</strong></td>
<td><strong>0.717</strong></td>
<td><strong>15.00</strong></td>
<td>81.27</td>
<td>0.739</td>
<td>129.70</td>
</tr>
<tr>
<td>MultiV+</td>
<td>74.14</td>
<td>0.698</td>
<td>23.33</td>
<td>77.67</td>
<td>0.728</td>
<td>190.31</td>
</tr>
<tr>
<td>MultiV+</td>
<td>74.20</td>
<td>0.699</td>
<td>14.69</td>
<td>74.03</td>
<td>0.698</td>
<td>122.65</td>
</tr>
<tr>
<td>MultiV+</td>
<td>75.33</td>
<td>0.703</td>
<td>21.85</td>
<td>74.23</td>
<td>0.699</td>
<td>192.38</td>
</tr>
<tr>
<td>MultiV+</td>
<td>67.81</td>
<td>0.658</td>
<td>14.25</td>
<td>65.42</td>
<td>0.648</td>
<td>125.40</td>
</tr>
<tr>
<td>MultiV+</td>
<td>69.59</td>
<td>0.661</td>
<td>21.05</td>
<td>63.14</td>
<td>0.644</td>
<td>187.00</td>
</tr>
</tbody>
</table>

Table 4.8: Experiments in the noisy case on real datasets. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.
Figure 4.27: Accuracy of the experiments in the noisy case on real datasets
Figure 4.28: NMI of the experiments in the noisy case on real datasets
Figure 4.29: Accuracy of the experiments in the noisy case on real datasets
Figure 4.30: NMI of the experiments in the noisy case on real datasets
**Chapter 5**

**Conclusions**

In this thesis project we presented multiple methods for community detection in multi-layer graphs. Most of the current algorithms either reduce multi-layer networks into a single-layer network or extend the algorithms for single-layer networks by using consensus clustering. However, these algorithms are criticized for their low accuracy because they either cannot preserve the community structure in compressed networks or ignore the connection among various layers. To overcome these problems, we tried to simultaneously take into account multiple layers.

The algorithms that we proposed are based on the Louvain heuristic method, that is a popular algorithm for community detection in single-layer graphs. It is an iterative procedure composed by two phases. During phase 1, it uses as criterion the modularity function, that is the most popular quality function for measuring the goodness of partitions. The most intuitive idea to extend this work to the multi-layer case, already studied in the literature, is to use the average of modularity on the layers.

The first method that we proposed, called Louvain Expansion, instead of considering just the modularity average, it takes into account the modularity variance.

The second algorithm, called Louvain Multiobjective, is more sophisticated and it is a filter type method, in fact it maintains just the modularity vectors that are not dominated according to a suitably developed Pareto search.

We suggested different versions of these methods to better analyse two situations: the informative case, where each layer has the same community structure, and the noisy case, where instead just some layers present a community structure and all the others are noise.

We implemented all the algorithms using Matlab. The methods have been tested on both artificial and real world networks. In both cases, we studied the informative and the noisy situation. We compared the results of the different algorithms calculating the accuracy and the Normalized Mutual Information.

We generated artificial networks with 2 and 3 layers using the Stochastic Block Model. In the informative case all the algorithms behave almost the same way unless in few occasions; instead in the noisy case the method already proposed in the literature, that considers just the average...
of modularity on the layers, is outdated by almost all the other methods. In particular, the best performances are achieved by the algorithms that consider the modularity variance as well as the modularity average giving them the same weight.

We tested our methods also on three real datasets from the literature. Also in this case we verified that our proposed approaches are competitive with the already proposed method. For time reasons, we did not test our models against state-of-the-art algorithms and this would be an additional step for providing robust results.

For further research in future studies, we could improve the models using a filter also in the phase 2 and trying to nominate the communities in different ways, s.a. randomizing the labels at the beginning of phase 1.

We should perform an in depth computational analysis with different values of $h$ length of the list and $\gamma$ weight of variance in the objective function.

A further goal is to test the methods on different type networks. For instance, we could use artificial graphs with different number of nodes, communities of various size, or with more than 3 layers. We should also test the algorithms on real networks from different fields.

A further idea is to evaluate and compare the outputs of the methods through the use of different tools.
Appendix A

Matlab code of the community-average method (Section 3.1)

% community-average
% Inputs :
% M : adjacent matrix
% s : 1 = Recursive computation
% : 0 = Just one level computation
% Output :
% COMTY, structure with the following information
% for each level i :
% COMTY.COM{i} : vector of community partition
% COMTY.SIZE{i} : vector of community sizes
% COMTY.MOD{i} : vector of modularities of clustering on the layers
% COMTY.Average(i) : average of modularity on the layers
% COMTY.Niter(i) : Number of iteration before convergence

function [COMTY ending] = community_average(M,z)
if nargin < 1
    error('not enough argument');
end
if nargin < 2
    z = 1;
end
S = size(M);
N = S(1);
if length(S)==3


k = S(3);
else k = 1;
end

ending = 0;

for s=1:k
    M2(:,:,s) = M(:,:,s) - diag(diag(M(:,:,s)));
end

% total weight of the graph
for s=1:k
    if z==1
        M(:,:,s) = M(:,:,s) + diag(diag(M(:,:,s)));
    end
    m{s} = sum(sum(M(:,:,s)))/2;
end

Niter = 0; % number of iterations
if (sum(cellfun(@(x)x>0,m))==0) | N == 1
    ending = 1;
    return
end

% Delate layers with m=0
M(:,:,cellfun(@(x)x==0,m))=[];
k = k - sum(cellfun(@(x)x==0,m));

COM = 1:S(1); % Community of node i
for s=1:k
    K(:,:,s) = sum(M(:,:,s)); % K(i)= sum of weight incident to node i
    SumTot(:,:,s) = sum(M(:,:,s)); % SumTot(i)= sum of weight incident to nodes in community i
    SumIn(:,:,s) = diag(M(:,:,s))'; % SumIn(i)= sum of weight inside community i (loops)
    Q{s} = compute_modularity(COM,M(:,:,s));
end
Average = sum([Q{:}])/k;

% Neighbor{j}{s} neighbor of node j in layer s
for j=1:N
    for s=1:k
        temp=M2(:,:,s);
        Neighbor_j{s} = find(temp(j,:));
    end
end
neighbor{\text{j}} = \text{Neighbor}_j;
end

gain = 1;
while (gain == 1)

gain = 0;
for i=1:N

\text{Ci} = \text{COM}(i);
\text{NB} = \text{unique(cat}(2,\text{Neighbor}(i){:})));
\text{G} = \text{zeros}(1,N); \% \text{Gain vector}
\text{best_increase} = -inf;
\text{Cnew} = \text{Ci};
\text{COM}(i) = -1;
\%remove \text{i} from its community
for s=1:k

\text{CNi} = (\text{COM}==\text{Ci}); \% \text{list of nodes in} \text{Ci community, without} \text{i}
\text{Ki_in_i\{s\}} = \text{sum} (\text{M}(i,\text{CNi},s)); \% \text{sum of weights between} \text{i} and \text{Ci}
\% \text{Gain of modularity}
\text{GQ_i\{s\}} = \text{K}(;,:s)\text{*SumTot}(;,:s,s)/(2*(m{s}^2))\text{ -} \text{Ki_in_i\{s\}}/m{s} - (((\text{K};,:s^2)/(2))/((2)))\text{ -} ((2)^2)/(2*(m{s}^2))^2);
\% \text{Recalculate values}
\text{SumTot}(;,:s,s) = \text{SumTot}(;,:s,s)\text{-} \text{K}(;,:s); \% \text{weights incident to}
\text{Ci community}
\text{SumIn}(;,:s,s) = \text{SumIn}(;,:s,s)\text{-} 2*\text{sum}(\text{M}(i,\text{CNi},s))\text{-} \text{M}(i,i,s); \% \text{weights community} \text{i}
\text{end}
\text{for} j=1:length(NB)

\text{Cj} = \text{COM}(\text{NB}(j));
\text{if} (\text{G}(\text{Cj}) == 0)

\text{CNj} = (\text{COM}==\text{Cj}); \% \text{nodes in community} \text{Cj, without} \text{j}
for s=1:k

\text{Ki_in_j\{s\}} = \text{sum} (\text{M}(i,\text{CNj},s)); \% \text{sum of weights between} \text{i} and \text{Cj}
\text{GQ_j\{s\}} = \text{Ki_in_j\{s\}}/m{s} - ((\text{K};,:s)\text{*SumTot}(;,:s,s))/(2*(m{s}^2)); \% \text{gain} \text{deltaQ} \text{if I put} \text{isolated node} \text{i in} \text{Cj}
\text{end}
\% \text{gain average}
\text{for} s=1:k

\text{DQ_j\{s\}} = \text{GQ_i\{s\}} + \text{GQ_j\{s\}};
\text{end}
\text{G}(\text{Cj}) = (\text{sum} ([\text{DQ_j}\{:\}])/k);
\text{if} \text{G}(\text{Cj}) > \text{best_increase}

\text{best_increase} = \text{G}(\text{Cj});
\text{DQ_j_t = DQ_j;}
\text{Cnew_t = Cj;}
\text{end}
\text{end}
\text{end}
if best_increase > -10^(-15)
    Cnew = Cnew_t;
    Average = Average + best_increase;
    for s=1:k
        Q{s} = Q{s} + DQ_j_t{s};
    end
end

% Recalculate
Ck = (COM==Cnew);
for s=1:k
    SumIn(:,Cnew,s) = SumIn(:,Cnew,s) + 2*sum(M(i,Ck,s)) + M(i,i,s);
    SumTot(:,Cnew,s) = SumTot(:,Cnew,s) + K(:,i,s);
end
COM(i) = Cnew;
if (Cnew ~= Ci)
    gain = 1;
end
Niter = Niter + 1;
end
Niter = Niter - 1;

if (z == 1)
    Mnew = M;
    Mold = Mnew;
    COMcur = COM;
    COMfull = COM;
    j = 2;
    while 1
        Mold = Mnew;
        S2 = size(Mold);
        Nnode = S2(1);
        COMu = unique(COMcur);
        Ncom = length(COMu);
        ind_com = sparse(Ncom,Nnode);
        ind_com_full = sparse(Ncom,N);
        for p=1:Ncom
            ind = find(COMcur==p);
            ind_com(p,1:length(ind)) = ind;
        end
        for p=1:Ncom
ind = find(COMfull==p);
ind_com_full(p,1:length(ind)) = ind;
end
Mnew = [];
for s=1:k
    Mnew(:,s,:) = zeros(Ncom,Ncom); % new matrix (each node is a community)
    for m=1:Ncom
        for n=m:Ncom
            ind1 = ind_com(m,:);
            ind2 = ind_com(n,:);
            % weights of edges between communities
            Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)));
            Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)));
        end
    end
end
[COMte] = community_average(Mnew,0);
if (e ~= 1)
    COMfull = sparse(1,N);
    COMcur = COMt.COM{1};
    for p=1:Ncom
        ind1 = ind_com_full(p,:);
        COMfull(ind1(ind1>0)) = COMcur(p);
    end
    [COMfull2] = reindex_com(COMfull);
    COMTY.COM{j} = COMfull2;
    COMTY.MOD{j} = COMt.MOD{1};
    COMTY.Average(j) = COMt.Average(1);
    COMTY.Niter(j) = COMt.Niter;
    Ind = (COMfull2 == COMTY.COM{j-1});
    if (sum(Ind) == length(Ind))
        return;
    end
else
    return;
end
j = j + 1;
end
end

% Re-index community partition by size
function [C Ss] = reindex_com(COMold)
C = sparse(1,length(COMold));
COMu = unique(COMold);
S = sparse(1,length(COMu));
for l=1:length(COMu)
    S(l) = length(COMold(COMold==COMu(l))); end
[Ss INDs] = sort(S,'descend');
for l=1:length(COMu)
    C(COMold==COMu(INDs(l))) = l;
end
end

%Compute modularity
function MOD = compute_modularity(C,Mat)
S = size(Mat);
N = S(1);
m = sum(sum(Mat))/2; %total weight
MOD = 0;
COMu = unique(C);
for j=1:length(COMu)
    Cj = (C==COMu(j)); %list of nodes in Cj
    Ec = sum(sum(Mat(Cj,Cj))); %sum of weights between nodes in Cj
    Et = sum(sum(Mat(Cj,:))); %sum of weights of nodes incident in nodes of Cj
    if Et>0
        MOD = MOD + Ec/(2*m)-(Et/(2*m))^2;
    end
end
end
Appendix B

Matlab code of the community-variance-minus method (Section 3.1)

```matlab
% community-variance-minus
% Inputs:
% M : weight matrix
% lambda : weight of variance in function for cut filter
% z : 1 = Recursive computation
% : 0 = Just one level computation
% Output:
% OUTPUT, structure with the following information
% for each level i :
% OUTPUT.COM{i} : vector of community partition
% OUTPUT.SIZE{i} : vector of community sizes
% OUTPUT.MOD{i} : vector of modularities of clustering on the layers
% OUTPUT.Average(i) : average of modularity on the layers
% OUTPUT.Niter(i) : Number of iteration before convergence

function [OUTPUT ending] = community_variance_minus(M, lambda)

if nargin < 1
    error('not enough argument');
end

if nargin < 2
    error('not lambda defined');
end

if nargin < 3
```
z = 1;
end

S = size(M);
N = S(1);
if length(S)==3
    k = S(3);
else k = 1;
end

ending = 0;
for s=1:k
    M2(:, :, s)=M(:, :, s)-diag(diag(M(:, :, s)));
end

% total weight of the graph
for s=1:k
    if z==1
        M(:, :, s) = M(:, :, s) + diag(diag(M(:, :, s)));
    end
    m{s} = sum(sum(M(:, :, s)))/2;
end

Niter = 0; % number of iterations
if (sum(cellfun(@(x)x >0 ,m))==0) | N == 1
    ending = 1;
    COMTY = 0;
    return;
end

% Delate layers with m=0
M(:, :, (cellfun(@(x)x==0 ,m)))=[];
k = k - sum(cellfun(@(x)x==0 ,m));

COM = 1:S(1); % Community of node i
for s=1:k
    K(:, :, s) = sum(M(:, :, s)); % K(i)= sum of weight incident to node i
    SumTot(:, :, s) = sum(M(:, :, s)); % SumTot(i)= sum of weight incident to nodes in community i
    SumIn(:, :, s) = diag(M(:, :, s))'; % SumIn(i)= sum of weight inside community i (loops)
    Q{s} = compute_modularity(COM,M(:, :, s)); % At the beginning each node is a community
end
Average = sum([Q{:}])/k;
if k == 1
    Variance = 0;
else
    Variance = (sum(([Q{:}] - Average).~(2))) / (k-1);  \%variance
end

Function = (1-lambda) * Average - lambda * Variance;  \%function

%Neighbor\{j\}{s} neighbor of node j in layer s
for j=1:N
    for s=1:k
        temp=M2(:,s);
        Neighbor_j{s} = find(temp(j,:));
    end
    Neighbor\{j\}=Neighbor_j;
end

gain = 1;
while (gain == 1)
    gain = 0;
    for i=1:N
        Ci = COM\{i\};
        NB=unique(cat(2,Neighbor\{i\}{s}));
        G = zeros(1,N);  \% Gain vector
        best_increase = -inf;
        Cnew = Ci;
        COM\{i\} = -1;
        \% remove i from its community
        for s=1:k
            CNi = (COM==Ci);  \%list of nodes in Ci community, without i
            Ki_in_i{s} = sum(M(i,CNi,s));  \%sum of weights between i and Ci
            \%Gain of modularity
            GQ_i{s} = K(:,i,s)*SumTot(:,Ci,s)/(2*(m{s}~(2))) - Ki_in_i{s}/m{s} - ((K(:,i,s))~(2))/(2*(m{s}~(2)));
            \%Recalculate values
            SumTot(:,Ci,s) = SumTot(:,Ci,s) - K(:,i,s);  \%weights incident to Ci community
            SumIn(:,Ci,s) = SumIn(:,Ci,s) - 2*sum(M(i,CNi,s)) - M(i,i,s);  \%weights community i
        end
        for j =1:length(NB)
            Cj = COM\{j\};
            if (G(Cj) == 0)
                CNj = (COM==Cj);  \%nodes in community Cj, without j
                for s=1:k
                    Ki_in_j{s} = sum(M(i,CNj,s));  \%sum of weights between i and Cj
                    GQ_j{s} = Ki_in_j{s}/m{s} - ((K(:,i,s)*SumTot(:,Cj,s))/(2*(m{s}~(2))));  \%gain deltaQ if I put isolated node i in Cj
                end
            end
            if (G(Cj) == 0)
                CNj = (COM==Cj);  \%nodes in community Cj, without j
                for s=1:k
                    Ki_in_j{s} = sum(M(i,CNj,s));  \%sum of weights between i and Cj
                    GQ_j{s} = Ki_in_j{s}/m{s} - ((K(:,i,s)*SumTot(:,Cj,s))/(2*(m{s}~(2))));  \%gain deltaQ if I put isolated node i in Cj
    end
        
end
% variance gain
if k == 1
    GV_j = 0;
else
    for s = 1:k
        DQ{s} = GQ_i{s} + GQ_j{s}; % gains
    end
    M_DQ = sum(DQ{k})/k; % gain average
    GV_j = ((sum(((DQ{k}) - M_DQ).^2) / (k-1)) + (2/(k-1)) * sum((sum(((Q{k}) - Average).^2) .* ((DQ{k}) - M_DQ))) % gain variance
end

% gain of the function
G(Cj) = (1 - lambda)*M_DQ - lambda*GV_j;
if G(Cj) > best_increase
    best_increase = G(Cj); % gain function
    Q_t = DQ; % gain of modularity;
    M_t = M_DQ; % gain average
    V_t = GV_j; % gain of variance
    Cnew_t = Cj;
end
end

if best_increase > -10^(-15)
    Cnew = Cnew_t;
    for s = 1:k
        Q{s} = Q{s} + Q_t{s};
    end
    Average = Average + M_t;
    Variance = Variance + V_t;
    Function = Function + best_increase;
end

% Recalculate
for s = 1:k
    COM(i) = Cnew;
end
if Cnew == Ci
    gain = 1;
end
Niter = Niter + 1;
end
Niter = Niter - 1;
[COM] = reindex_com(COM);
COMTY.COM{1} = COM;
COMTY.MOD{1} = [Q{:}];
COMTY.Average(1) = Average;
COMTY.Variance(1) = Variance;
COMTY.Function(1) = Function;
COMTY.Niter(1) = Niter;

% Perform part 2
if (z == 1)
    Mnew = M;
    Mold = Mnew;
    COMcur = COM;
    COMfull = COM;
    j = 2;

    while 1
        Mold = Mnew;
        S2 = size(Mold);
        Nnode = S2(1);

        COMu = unique(COMcur);
        Ncom = length(COMu);
        ind_com = sparse(Ncom,Nnode);
        ind_com_full = sparse(Ncom,N);
        for p=1:Ncom
            ind = find(COMcur==p);
            ind_com(p,1: length(ind)) = ind;
        end
        for p=1:Ncom
            ind = find(COMfull==p);
            ind_com_full(p,1:length(ind)) = ind;
        end
        Mnew = [];
        for s=1:k
            Mnew(:,:,s) = zeros(Ncom,Ncom); % new matrix (each node is a community)
            for m=1:Ncom
                for n=m:Ncom
                    ind1 = ind_com(m,:);
                    ind2 = ind_com(n,:);
                    % weights of edges between communities
                    Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)));
                    Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)));
                end
            end
        end
        [COMt e] = community_variance_minus(Mnew, lambda,0);
        if (e ~= 1)
            COMfull = sparse(1,N);
        end
    end
end
COMcur = COMt.COM{1};
for p=1:Ncom
    ind1 = ind_com_full(p,:);
    COMfull(ind1(ind1>0)) = COMcur(p);
end
[COMfull2] = reindex_com(COMfull);
COMTY.COM{j} = COMfull2;
COMTY.MOD{j} = COMt.MOD{1};
COMTY.Average(j) = COMt.Average(1);
COMTY.Variance(j) = COMt.Variance(1);
COMTY.Function(j) = COMt.Function(1);
COMTY.Niter(j) = COMt.Niter;
Ind = (COMfull2 == COMTY.COM{j-1});
if (sum(Ind) == length(Ind))
    return;
else
    return;
end
j = j + 1;
end
end

% Re-index community partition by size
function [C Ss] = reindex_com(COMold)
C = sparse(1,length(COMold));
COMu = unique(COMold);
S = sparse(1,length(COMu));
for l=1:length(COMu)
    S(l) = length(COMold(COMold == COMu(l)));
end
[Ss INDs] = sort(S,'descend');
for l=1:length(COMu)
    C(COMold==COMu(INDs(l))) = l;
end
end

%Compute modularity
function MOD = compute_modularity(C,Mat)
S = size(Mat);
N = S(1);
m = sum(sum(Mat))/2; %total weight
MOD = 0;
COMu = unique(C);
for j=1:length(COMu)
    Cj = (C==COMu(j)); %list of nodes in Cj
    Ec = sum(sum(Mat(Cj,Cj))); %sum of weights between nodes in Cj
Et = sum(sum(Mat(Cj,:))); % sum of weights of nodes incident in nodes of Cj

if Et > 0
    MOD = MOD + Ec/(2*m) - (Et/(2*m))^2;
end

end

end
Appendix C

Matlab code of the community-variance-plus method (Section 3.1)

```matlab
% community-variance-plus
% Inputs :
% M : weight matrix
% lambda : weight of variance in function for cut filter
% z : 1 = Recursive computation
% : 0 = Just one level computation
% Output :
% COMTY, structure with the following information
% for each level i :
% COMTY.COM{i} : vector of community partition
% COMTY.SIZE{i} : vector of community sizes
% COMTY.MOD{i} : vector of modularities of clustering on the layers
% COMTY.Average(i) : average of modularity on the layers
% COMTY.Niter(i) : Number of iteration before convergence
%
function [COMTY ending] = community_variance_plus(M,lambda,z)

if nargin < 1
    error('not enough argument');
end

if nargin < 2
    error('not lambda defined');
end

if nargin < 3
```
z = 1;
end

S = size(M);
N = S(1);
if length(S)==3
    k = S(3);
else k = 1;
end

ending = 0;

for s=1:k
    M2(:,:,s)=M(:,:,s)-diag(diag(M(:,:,s)));% total weight of the graph
end

for s=1:k
    if z==1
        M(:,:,s) = M(:,:,s) + diag(diag(M(:,:,s)));
    end
    m{s} = sum(sum(M(:,:,s))) / 2;
end

Niter = 0; % number of iterations
if (sum(cellfun(@(x)x>0,m))==0) | N == 1
    ending = 1;
    COMTY = 0;
    return;
end

% Delate layers with m=0
M(:,:,cellfun(@(x)x==0,m))=[];
k = k - sum(cellfun(@(x)x==0,m));

COM = 1:S(1); % Community of node i
for s=1:k
    K(:,:,s) = sum(M(:,:,s)); % K(i)= sum of weight incident to node i
    SumTot(:,:,s) = sum(M(:,:,s)); %SumTot(i)= sum of weight incident to nodes in community i
    SumIn(:,:,s) = diag(M(:,:,s))'; %SumIn(i)= sum of weight inside community i (loops)
    Q{s} = compute_modularity(COM,M(:,:,s));%At the beginning each node is a community
end
Average = sum([Q{:}])/k;
if k==1
    Variance = 0;
else
    Variance = (sum(((Q{:})-Average).^2)) / (k-1); %variance
end

Function = (1-lambda) * Average + lambda * Variance; %function

%Neighbor{j}{s} neighbor of node j in layer s
for j=1:N
    for s=1:k
        temp=M2(:,s);
        Neighbor_j{s} = find(temp(j,:));
    end
    Neighbor{j}=Neighbor_j;
end

gain = 1;
while (gain == 1)
    gain = 0;
    for i=1:N
        Ci = COM(i);
        NB=unique(cat(2,Neighbor{i}{:}));
        G = zeros(1,N); % Gain vector
        best_increase = -inf;
        Cnew = Ci;
        % remove i from its community
        for s =1: k
            CNi = (COM==Ci); %list of nodes in Ci community, without i
            Ki_in_i{s} = sum(M(i,CNi,s)); %sum of weights between i and Ci
            %Gain of modularity
            GQ_i{s} = K(:,i,s)*SumTot(:,Ci,s)/(2*(m{s}^2)) - Ki_in_i{s}/m{s} - ((K(:,i,s))^2)/(2*(m{s}^2));
            %Recalculate values
            SumTot(:,Ci,s) = SumTot(:,Ci,s) - K(:,i,s); %weights incident to Ci community
            SumIn(:,Ci,s) = SumIn(:,Ci,s) - 2*sum(M(i,CNi,s)) - M(i,i,s); %weights community i
        end
        for j =1:length(NB)
            Cj = COM(NB(j));
            if (G(Cj) == 0)
                CNj = (COM==Cj); %nodes in community Cj, without j
                for s=1:k
                    Ki_in_j{s} = sum(M(i,CNj,s)); %sum of weights between i and Cj
                    GQ_j{s} = Ki_in_j{s}/m{s} - (K(:,i,s)*SumTot(:,Cj,s))/(2*(m{s}^2)); %gain deltaQ if I put isolated node i in Cj
                end
            end
        end
    end
end
if k==1
    GV_j = 0;
else
    for s=1:k
        DQ{s} = GQ_i{s} + GQ_j{s}; % gains
    end
    M_DQ = sum([DQ{:}])/k; % gain average
    GV_j = ((sum(([DQ{:}] - M_DQ).^2)) / (k-1)) + (2/(k-1)) * sum(sum(([Q{:}] - Average) .* ([DQ{:}] - M_DQ))); % gain variance
end

% gain of the function
G(Cj) = (1 - lambda)*M_DQ + lambda* GV_j;
if G(Cj) > best_increase
    best_increase = G(Cj); % gain function
    Q_t = DQ; % gain of modularity;
    M_t = M_DQ; % gain average
    V_t = GV_j; % gain of variance
    Cnew_t = Cj;
end
if best_increase > -10^(-15)
    Cnew = Cnew_t;
    for s=1:k
        Q{s} = Q{s} + Q_t{s};
    end
    Average = Average + M_t;
    Variance = Variance + V_t;
    Function = Function + best_increase;
end
% Recalculate
Ck = (COM==Cnew);
for s=1:k
    SumIn(:,Cnew,s) = SumIn(:,Cnew,s) + 2*sum(M(i,Ck,s)) + M(i,i,s);
    SumTot(:,Cnew,s) = SumTot(:,Cnew,s) + K(:,i,s);
end
COM(i) = Cnew;
if (Cnew ~= Ci)
    gain = 1;
end
end
Niter = Niter + 1;
end
Niter = Niter - 1;
[COM] = reindex_com(COM);
COMTY.COM{1} = COM;
COMTY.MOD{1} = [Q{:}];
COMTY.Average(1) = Average;
COMTY.Variance(1) = Variance;
COMTY.Function(1) = Function;
COMTY.Niter(1) = Niter;

% Perform part 2
if (z == 1)
    Mnew = M;
    Mold = Mnew;
    COMcur = COM;
    COMfull = COM;
    j = 2;

while 1
    Mold = Mnew;
    S2 = size(Mold);
    Nnode = S2(1);
    COMu = unique(COMcur);
    Ncom = length(COMu);
    ind_com = sparse(Ncom,Nnode);
    ind_com_full = sparse(Ncom,N);
    for p=1:Ncom
        ind = find(COMcur==p);
        ind_com(p,1:length(ind)) = ind;
    end
    for p=1:Ncom
        ind = find(COMfull==p);
        ind_com_full(p,1:length(ind)) = ind;
    end
    Mnew = [];
    for s=1:k
        Mnew(:,s) = zeros(Ncom,Ncom); % new matrix (each node is a
                                      % community)
        for m=1:Ncom
            for n=m:Ncom
                % weights of edges between communities
                Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)
                                      ));
                Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)
                                      ));
            end
        end
    end
    [COMt e] = community_variance_plus(Mnew,lambda,0);
    if (e ~= 1)
        COMfull = sparse(1,N);
    end
end
for p=1:Ncom
  ind1 = ind_com_full(p,:);
  COMfull(ind1(ind1>0)) = COMcur(p);
end
[COMfull2] = reindex_com(COMfull);
COMTY.COM{j} = COMfull2;
COMTY.MOD{j} = COMt.MOD{1};
COMTY.Average(j) = COMt.Average(1);
COMTY.Variance(j) = COMt.Variance(1);
COMTY.Function(j) = COMt.Function(1);
COMTY.Niter(j) = COMt.Niter;
Ind = (COMfull2 == COMTY.COM{j-1});
if (sum(Ind) == length(Ind))
  return;
else
  return;
end
j = j + 1;
end
end

% Re-index community partition by size
function [C Ss] = reindex_com(COMold)
C = sparse(1,length(COMold));
COMu = unique(COMold);
S = sparse(1,length(COMu));
for l=1:length(COMu)
  S(l) = length(COMold(COMold == COMu(l)));
end
[Ss INDs] = sort(S,'descend');
for l=1:length(COMu)
  C(COMold==COMu(INDs(l))) = l;
end
end

% Compute modularity
function MOD = compute_modularity(C,Mat)
S = size(Mat);
N = S(1);
m = sum(sum(Mat))/2; % total weight
MOD = 0;
COMu = unique(C);
for j=1:length(COMu)
  Cj = (C==COMu(j)); % list of nodes in Cj
\[ Ec = \text{sum} (\text{sum} (\text{Mat} (C_j, C_j))); \text{ sum of weights between nodes in } C_j \]

\[ Et = \text{sum} (\text{sum} (\text{Mat} (C_j, :))); \text{ sum of weights of nodes incident in nodes of } C_j \]

\[ \text{if } Et > 0 \]

\[ \text{MOD} = \text{MOD} + Ec/(2* \text{m}) - (Et/(2* \text{m}))^-2; \]

\[ \text{end} \]

\[ \text{end} \]
Appendix D

Matlab code of the multi-average method (Section 3.2)

```matlab
% multi-average
% Inputs :
% M : vector of weight matrix of each layer
% h : length of the filter
% z : 1 = Recursive computation
% : 0 = Just one level computation
% Output :
% L the filter with the following information
% for each element l:
% l{1}=Q cell with modularity of each layer
% l{2}={COMcur COMfull COMindex} communities in the current graph, in
% the original graph, communities in the current graph reindexed
% l{3}=SumTot (sum of weights of the links incident to nodes in a
% community)
% l{4}=SumIn (sum of weights of the links inside a community)
% l{5}=Average (average)

function [L ending] = multi_average(M,h,z)
if nargin < 1
    error ('not enough argument');
end
if nargin < 2
    error ('not h defined');
end
if nargin < 3
    z = 1;
end
```
S = size(M);
N = S(1);
if length(S)==3
    k = S(3);
else k = 1;
end

ending = 0;

for s = 1:k
    M2(:, :, s) = M(:, :, s) - diag(diag(M(:, :, s)));
end

% total weight of the graph
for s = 1:k
    if z == 1
        M(:, :, s) = M(:, :, s) + diag(diag(M(:, :, s)));
    end
    m{s} = sum(sum(M(:, :, s))) / 2;
end

Niter = 0; % number of iterations

% Calculation of the beginning values
COM{1} = 1:S(1); % current community % At the beginning each node is a community
COM{2} = 1:S(1); % original graph community
COM{3} = 1:S(1); % community reindexed
% COM(i)= Community of node i

for s = 1:k
    K(:, :, s) = sum(M(:, :, s)); % K(i)= sum of weight incident to node i
    SumTot(:, :, s) = sum(M(:, :, s)); % SumTot(i)= sum of weight incident to nodes in community i
    SumIn(:, :, s) = diag(M(:, :, s))'; % SumIn(i)= sum of weight inside community i (loops)
    % At the beginning each node is a community
    Q{s} = compute_modularity(COM{1}, M(:, :, s));
end

% average
Average = sum([Q{:}]) / k;

% filter
L={[Q],{COM},SumTot,SumIn,Average};

% If no edges in any layer or just one node
if (sum(cellfun(@(x)x>0,m))==0) | N == 1 %| logical or
ending = 1;
else
    
    % Delete layers with m=0
    M(:, :, (cellfun(@(x)x==0, m)) == []) = [];
    k = k - sum(cellfun(@(x)x==0, m));
    
    % Neighbor {j}{s} neighbor of node j in layer s
    for j = 1:N
        for s = 1:k
            temp = M2(:, :, s);
            Neighbor_j{s} = find(temp(:, :));
        end
        Neighbor{j} = Neighbor_j;
    end
    
    GAIN = 1;
    while (GAIN == 1) % Stop when putting nodes in other communities do not
    increase the modularity
        L_old = L;
        L_new = L;
        for i = 1:N % for each node, in this order
            L_o = L;
            for l = 1:length(L_o) % for each situation in the filter
                % delete the point from the filter
                index = cellfun(@(x) isequal(x, L_o{l}), L_new, 'UniformOutput',
                                true, 1);
                L_new(index) = [];
                % step 1
                [L_new, U] = step_1({L_o{l}{1}}, {L_o{l}{2}}, L_o{l}{3}, L_o{l}
                                {4}, L_o{l}{5}, k, Neighbor, K, N, m, L_new, h, i);
                % if no new point, insert again the initial point in the filter
                if U == 0
                    L_new = L_new_o;
                end
            end
            L_new = cut_filter(L_new, h, k);
            L = L_new;
        end
        % Check if nothing happened
        if length(L_old) == length(L_new)
            GAIN = 1;
        else
            C_old = [];
            C_new = [];
            for l = 1:length(L_old)
                R_old = [];
                end
            else
R_new=[];
for s=1:k
    R_old = [R_old L_old{l}{1}{s}];
    R_new = [R_new L_new{l}{1}{s}];
end
C_old = [C_old ; R_old];
C_new = [C_new ; R_new];
end
C = ismembertol(C_old,C_new,10^(-6));
if size(C,1)*size(C,2) == sum(sum(C))
    GAIN=0;
end
Niter = Niter + 1;
end

% Perform part 2
if (z==1)
    L = cut_filter(L,1,k); % one final element
    Mnew=M; % new weight matrix of this iteration
    Mold=Mnew; % old weight matrix of iteration before
    COMcur = L{l}{2}{3}; % communities in the graph of this iteration
    COMfull = L{l}{2}{3}; % communities in the original graph
    LL=L;
    j=2;% number of pass (step1+step2)
    S=1;
    while S
        L_old=LL;
        Mold = Mnew;
        S2 = size(Mold); % number of nodes
        Nnode = S2(1);
        COMu = unique(COMcur); % list of communities without repetitions
                       and in sorted order
        Ncom = length(COMu); % number of communities
        ind_com = sparse(Ncom,Nnode); % zero matrix with a row for each
            community and a column for each node in this configuration
        ind_com_full = sparse(Ncom,N); % zero matrix with a row for each
            community and a column for each node of the original graph
        for p=1:Ncom % for each community
            ind = find(COMcur==p);
            ind_com(p,1:length(ind)) = ind;
        end
        for p=1:Ncom
            ind = find(COMfull==p);
            ind_com_full(p,1:length(ind)) = ind;
        end
end
Mnew=[];
for s=1:k
    Mnew(:,:,s) = sparse(Ncom,Ncom); % new matrix (each node is a community)
    for m=1:Ncom
        for n=m:Ncom
            ind1 = ind_com(m,:);
            ind2 = ind_com(n,:);
            % weights of edges between communities
            Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)));
            Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)));
        end
    end
end
% apply first step to this new matrix but z=0 without recursive
[LLL e] = multi_average(Mnew,h,0);
if isempty(LLL)
    L=LLL;
    return
else
    LL = cut_filter(LLL,1,k); % one final element
    %if (e ~= 1)
    COMfull = sparse(1,N); % communities of the original graph
    COMcur = LL{1}{2}{3}; % communities
    for p=1:Ncom
        ind1 = ind_com_full(p,:); % nodes in community p in the original graph
        COMfull(ind1(ind1>0)) = COMcur(p); % community now of node i
    end
    [COMfull] = reindex_com(COMfull);
    LL{1}{2}{2}=COMfull;
    % communities do not change
    if isequal(L_old{1}{2}{2},LL{1}{2}{2})
        % return not just this value, but all the value of the last list
        % calculate COMfull for each element of the last list
        for l=1:length(LLL)
            COMfull = sparse(1,N); % communities of the original graph
            COMcur = LL{l}{2}{3}; % communities
            for p=1:Ncom
                ind1 = ind_com_full(p,:); % nodes in community p in the original graph
                COMfull(ind1(ind1>0)) = COMcur(p); % community now of node i
            end
        end
    end
end
[COMfull] = reindex_com(COMfull);

LLL(1){2}{2}=COMfull;
end
L = LLL;
S=0;
end

j = j + 1; % start another pass

if tEnd > 1200
    L = {}; return
end
end
end
end
end
end

% Compute step_1

function [L,U] = step_1(Q,COM,SumTot,SumIn,Average,k,Neighbor,K,M,N,m,L,h,i)

U=0; % if U=1 add at least an element to the list, otherwise put again the initial point in the filter

COMcur=COM{1}{1}; % current community
Ci = COMcur(i); % community of node i

NB=unique(cat(2,Neighbor{i}{:}));

SumTot2=SumTot;
SumIn2=SumIn;

for s=1:k

    COMcur2 = COMcur;
    COMcur2(i) = -1;
    CNi = (COMcur2==Ci); % list of nodes in Ci community, without i
    Ki_in_i{s} = sum(M(i,CNi,s)); % sum of weights between i and Ci
    GQ_i{s} = (K(:,i,s)*SumTot2(:,Ci,s))/(2*(m{s}^2)) - Ki_in_i{s}/m{s} - ((K(:,i,s))^(2))/(2*(m{s}^2));

    SumTot2(:,Ci,s) = SumTot2(:,Ci,s) - K(:,i,s); % weights incident to Ci community
    SumIn2(:,Ci,s) = SumIn2(:,Ci,s) - 2*sum(M(i,CNi,s)) - M(i,i,s); % weights community i

end

G = sparse(1,N);
for j=1:length(NB)
    SumTot3=SumTot2;

SumIn3 = SumIn2;
Cj = COMcur(NB(j)); %community of node j
if (G(Cj) == 0) %If not tried with another node of community Cj yet
    G(Cj) = 1;
    for s = 1:k
        COMcur3 = COMcur;
        COMcur3(i) = -1;
        % I put i in the community of j for each layer
        CNj = (COMcur3 == Cj); %nodes in community Cj, without j
        Ki_in_j{s} = sum(M(i,CNj,s)); %sum of weights between i and Cj
        GQ_j{s} = Ki_in_j{s}/m{s} - (K(:,i,s)*SumTot3(:,Cj,s))/(2*(m{s}^2)); %gain deltaQ if I put isolated node i in Cj
        % Recalculate
        SumTot3(:,Cj,s) = SumTot3(:,Cj,s) + K(:,i,s);
        SumIn3(:,Cj,s) = SumIn3(:,Cj,s) + 2*sum(M(i,CNj,s)) + M(i,i,s);
    end
    COMcur4 = COMcur;
    COMcur4(i) = Cj;
    COM{1}{1} = COMcur4;
    [COM{1}{3}] = reindex_com(COMcur4);
end
GA_j = sum([DQ{:}])/k; %gain average
if GA_j > -10^-14
    A_j = Average + GA_j;
    %modularity
    for s = 1:k
        Q_j{s} = Q{1}{s} + DQ{s};
    end
    [L] = add_to_filter({Q_j},COM,SumTot3,SumIn3,A_j,L,k);
    U = 1;
end
end
end
% add a point to the filter
function [L] = add_to_filter(QQ,COM,SumTot,SumIn,Average,L,k)
if length(L) ~= 0
    COMcur = COM{1}{1}; %current community
    %i. Check if the new point is dominated by a point in the filter
    DD = 0;
l=1;
while (DD==0) & l<=length(L)
  D=0;
  s=1;
  while (D==0) & s<=k
    if QQ{1}{s}>L{l}{1}{s}
      D=1;
    end
    s=s+1;
  end
  if D %the new point is not dominated by l
    DD=1;
  end
  l=l+1;
end
if DD %the new point is not dominated
  DDD=0;
  l=1;
  while (DDD==0) & l<=length(L)
    D=1;
    s=1;
    while D & s<=k
      if abs(L{l}{1}{s} - QQ{1}{s}) > 10^(-14)
        D=0;
      end
      s = s + 1;
    end
    if D
      DDD=1;
    end
    l = l + 1;
  end
  if DDD==0
    %2. Check if the other points in the filter are dominated by the new
    % point %
    for l=1:length(L)
      D=1;
      s=1;
      while D & s<=k
        if L{l}{1}{s}>QQ{1}{s}+10^(-14)
          D=0;
        end
        s=s+1;
      end
      if D %l is dominated by the new point so I remove it from the
        filter

108
L{1} = []; 

end 

end 

empties = (cellfun (@isempty, L)); 
L(empties) = []; 
L{end + 1} = [QQ, COM, SumTot, SumIn, Average]; % Add the new point to the filter 

end 

end 

else 
L{end + 1} = [QQ, COM, SumTot, SumIn, Average]; % Add the new point to the filter 
end 

end 

end 

end

% cut the filter
function [L] = cut_filter(L, h, k)

% calculate the function for every element of the list
if length(L) > h 
    for l = 1:length(L) 
        % function
        Average{l} = L{l}{5}; 
    end 

% remove the situations in the filter with the lower values
for t = 1:(length(L) - h) 
    [mn, idx] = min([Average{:}]); 
    v = find([Average{:}] == mn); % if more element with same average values, delete the last one (because with h=1 I have same result as cluster_jl_average_variance)
    idx_t = v(end);
    Average{idx_t} = Inf; 
    L{idx_t} = []; 
end 

empties = find (cellfun (@isempty, L)); 
L(empties) = []; 
end 

end

% Compute modularity
function MOD = compute_modularity(C, Mat) 

S = size(Mat); 
N = S(1);
\[ m = \frac{\text{sum} (\text{sum} (\text{Mat}))/2; \% \text{total weight} \]

\[ \text{MOD} = 0; \]

\[ \text{COMu} = \text{unique} (C); \% \text{list of communities without repetitions and in sorted order} \]

\[ \% \text{for each community calculate modularity and then sum all together} \]

\[ \text{for } j=1: \text{length} (\text{COMu}) \]

\[ \text{Cj} = (\text{C}==\text{COMu}(j)); \% \text{list of nodes in Cj} \]

\[ \% \text{faster then Cj} = \text{find} (\text{C}==\text{COMu}(j)) \]

\[ \text{Ec} = \text{sum} (\text{sum} (\text{Mat} (\text{Cj}, \text{Cj}))); \% \text{sum of weights between nodes in Cj} \]

\[ \% \text{Mat(Cj,Cj) submatrix} \]

\[ \text{Et} = \text{sum} (\text{sum} (\text{Mat} (\text{Cj}, :))); \% \text{sum of weights of nodes incident in nodes of Cj} \]

\[ \text{if Et}>0 \]

\[ \text{MOD} = \text{MOD} + \frac{\text{Ec}^{2}}{(2* m)} - \frac{(\text{Et}^{2})}{(2* m)^{2}}; \]

\[ \text{end} \]

\[ \text{end} \]

\[ \% \text{Re-index community partition by size} \]

\[ \text{function [C Ss] = reindex_com (COMold)} \]

\[ \text{C} = \text{sparse} (1, \text{length} (\text{COMold})); \]

\[ \text{COMu} = \text{unique} (\text{COMold}); \]

\[ \text{S} = \text{sparse} (1, \text{length} (\text{COMu})); \]

\[ \text{for } l=1: \text{length} (\text{COMu}) \]

\[ \text{S}(l) = \text{length} (\text{COMold} (\text{COMold} == \text{COMu}(l))); \]

\[ \text{end} \]

\[ \text{[Ss INDs]} = \text{sort} (\text{S}, ' \text{descend} '); \]

\[ \text{for } l=1: \text{length} (\text{COMu}) \]

\[ \text{C} (\text{COMold} == \text{COMu}(\text{INDs}(l))) = l; \]

\[ \text{end} \]

\[ \% \text{Re-index community partition no by size but by initial order} \]

\[ \% \text{function [C] = reindex_com (COMold)} \]

\[ \% \text{C} = \text{sparse} (1, \text{length} (\text{COMold})); \% \text{vector with a index for each node} \]

\[ \% \text{COMu} = \text{unique} (\text{COMold}); \% \text{unique(v)=same data of v but not repetitions and in sorted order} \]

\[ \% \text{COMolds} = \text{sort} (\text{COMold}); \]

\[ \% \text{COMu} = \text{COMolds} ([\text{true}; \text{diff} (\text{COMolds}(:))>0]); \]

\[ \% \text{for } l=1: \text{length} (\text{COMu}) \]

\[ \% \quad \text{C} (\text{COMold} == \text{COMu}(l)) = 1; \% \text{rename the communities} \]

\[ \% \text{end} \]

\[ \% \text{end} \]
Appendix E

Matlab code of the multi-variance-minus method (Section 3.2)

```matlab
% multi-variance-minus
% Inputs:
% M : vector of weight matrix of each layer
% z : 1 = Recursive computation
% : 0 = Just one level computation
% Output:
% L the filter with the following information
% for each element l:
% l{1}=Q cell with modularity of each layer
% l{2}={COMcur COMfull COMindex} communities in the current graph, in
% the original graph, communities in the current graph reindexed
% l{3}=SumTot (sum of weights of the links incident to nodes in a
% community)
% l{4}=SumIn (sum of weights of the links inside a community)
% l{5}=Average (average)
% l{6}=Function (funtion)
%
function [L ending] = multi_variance_minus(M,lambda,h,z)

if nargin < 1
    error('not enough argument');
end
if nargin < 2
    error('not lambda defined');
end
```
if nargin < 3
    error('not h defined');
end
if nargin < 4
    z = 1;
end
S = size(M);
N = S(1);
if length(S)==3
    k = S(3);
else
    k = 1;
end
ending = 0;
for s = 1:k
    M2(:, :, s) = M(:, :, s) - diag(diag(M(:, :, s)));
end
% total weight of the graph
for s = 1:k
    if z==1
        M(:, :, s) = M(:, :, s) + diag(diag(M(:, :, s)));
    end
    m{s} = sum(sum(M(:, :, s))) / 2;
end
Niter = 0; % number of iterations
% Calculation of the beginning values
COM{1} = 1:S(1); % current community % At the beginning each node is a community
COM{2} = 1:S(1); % original graph community
COM{3} = 1:S(1); % community reindexed
% COM(i)= Community of node i
for s = 1:k
    K(:, :, s) = sum(M(:, :, s)); % K(i)= sum of weight incident to node i
    SumTot(:, :, s) = sum(M(:, :, s)); % SumTot(i)= sum of weight incident to nodes in community i
    SumIn(:, :, s) = diag(M(:, :, s)); % SumIn(i)= sum of weight inside community i (loops)
    Q{s} = compute_modularity(COM{1}, M(:, :, s));
end
% average
Average = sum([Q{:}]) / k;

% variance
if k==1
    Variance = 0;
else
    Variance = (sum(([Q{:}]-Average).^2)) / (k-1);
end

% function
Function = (1-lambda) * Average - lambda * Variance;

% filter
L={[Q],{COM},SumTot,SumIn,Average,Function};

% If no edges in any layer or just one node
if (sum(cellfun (@(x)x >0 ,m)) ==0) | N == 1 %| logical or
    ending = 1;
else
    % Delate layers with m=0
    M(:,:,1)=cellfun(@(x)x==0 ,m));
    k = k - sum(cellfun(@(x)x==0 ,m));
    % Neighbor {j}{s} neighbor of node j in layer s
    for j =1: N
        for s =1: k
            temp =M(:,:,s);
            Neighbor_j {s} = find ( temp (j,:));
        end
        Neighbor {j}= Neighbor_j ;
    end

    GAIN = 1;
    while (GAIN == 1) % Stop when putting nodes in other communities do not
                    % increase the modularity
        L_old=L;
        L_new=L;
        for i =1: N % for each node, in this order
            L_o=L;
            for l =1:length(L_o) % for each situation in the filter
                % delate the point from the filter
                L_new_o=L_new;
                index = cellfun(@(x) isequal(x,L_o(l))), L_new, 'UniformOutput'
                , 1);
                L_new(index)=[];
                % step 1
                [L_new,U] = step_1 ([L_o{l}{1}],[L_o{l}{2}],[L_o{l}{3}],[L_o{l}
                {4}],[L_o{l}{5}],[L_o{l}{6}],k,Neighbor,K,M,N,m,L_new,h,lambda,
                i);
                % if no new point, put again the initial point in the filter
if U==0
    L_new=L_new_o;
end
L_new = cut_filter(L_new,h,k,lambda);
L=L_new;
end
%Check if nothing happened
if length(L_old)~=length(L_new)
    GAIN=1;
else
    C_old=[];
    C_new=[];
    for l=1:length(L_old)
        R_old=[];
        R_new=[];
        for s=1:k
            R_old = [R_old L_old{l}{1}{s}];
            R_new = [R_new L_new{l}{1}{s}];
        end
        C_old = [C_old ; R_old];
        C_new = [C_new ; R_new];
    end
    C = ismembertol(C_old,C_new,10^(-6));
    if size(C,1)*size(C,2) == sum(sum(C))
        GAIN=0;
    end
end
Niter = Niter + 1;
end

% Perform part 2
if (z==1)
    L = cut_filter(L,1,k); %one final element
    Mnew=M; %new weight matrix of this iteration
    Mold=Mnew; %old weight matrix of iteration before
    COMcur = L{1}{2}{3}; %communities in the graph of this iteration
    COMfull = L{1}{2}{3}; %communities in the original graph
    LL=L;
    j=2; %number of pass (step1+step2)
    S=1;
    while S
        L_old=LL;
        Mold = Mnew;
        S2 = size(Mold); %number of nodes
        Nnode = S2(1);
        COMu = unique(COMcur); %list of communities without repetitions
        end
end
and in sorted order
Ncom = length(COMu); % number of communities
ind_com = sparse(Ncom,Nnode); % zero matrix with a row for each
community and a column for each node in this configuration
ind_com_full = sparse(Ncom,N); % zero matrix with a row for each
community and a column for each node of the original graph

for p = 1:Ncom % for each community
    ind = find(COMcur == p);
    ind_com(p,1:length(ind)) = ind; % in row p put the indeces of
    % the nodes in community p in this configuration
end

for p = 1:Ncom
    ind = find(COMfull == p);
    ind_com_full(p,1:length(ind)) = ind; % in row p put the
    indeces of the nodes in community p in the original graph
end
Mnew = [];
for s = 1:k
    Mnew(:, :, s) = sparse(Ncom,Ncom); % new matrix (each node is
    % a community)
    for m = 1:Ncom
        for n = m:Ncom
            ind1 = ind_com(m, :);
            ind2 = ind_com(n, :);
            % weights of edges between communities
            Mnew(m,n,s) = sum(sum(Mold(ind1(ind1 > 0), ind2(ind2 > 0)
                                    , s)));
            Mnew(n,m,s) = sum(sum(Mold(ind1(ind1 > 0), ind2(ind2 > 0)
                                    , s)));
        end
    end
end
% apply first step to this new matrix but z=0 without recursive
[LLL e] = multi_variance_minus(Mnew, lambda, h, 0);
if isempty(LLL)
    L = LLL;
    return
else
    LL = cut_filter(LLL,1,k); % one final element
    COMfull = sparse(1,N); % communities of the original graph
    COMcur = LL{1}{2}{3}; % communities
    for p = 1:Ncom
        ind1 = ind_com_full(p, :); % nodes in community p in the
        original graph
        COMfull(ind1(ind1 > 0)) = COMcur(p); % community now of
        node i
    end
    [COMfull] = reindex_com(COMfull);
LL{1}{2}{2} = COMfull;
% communities do not change
if isequal(L_old{1}{2}{2}, LL{1}{2}{2})
% return not just this value, but all the value of the last list
% calculate COMfull for each element of the last list
for l = 1:length(LLL)
    COMfull = sparse(1, N); % communities of the original graph
    COMcur = LLL{1}{2}{3}; % communities
    for p = 1:Ncom
        ind1 = ind_com_full(p, :); % nodes in community p in the original graph
        COMfull(ind1(ind1 > 0)) = COMcur(p); % community now of node i
    end
    [COMfull] = reindex_com(COMfull);
    LLL{1}{2}{2} = COMfull;
end
end
L = LLL;
S = 0;
end
end
end
end
end
end
end
end
end
end
end
end
end
end
end
end
end
end
end  

% Compute step_1
function [L, U] = step_1(Q, COM, SumTot, SumIn, Average, Function, k, Neighbor, K, M, N, m, L, h, lambda, i)
U = 0; % if U=1 add at least an element to the list, otherwise insert again the initial point in the filter
COMcur = COM{1}{1}; % current community
Ci = COMcur(i); % community of node i
NB = unique(cat(2, Neighbor{i}{:}));
SumTot2 = SumTot;
SumIn2 = SumIn;
% remove i from its community
for s = 1:k

116
COMcur2 = COMcur;
COMcur2(i) = -1;
CNi = (COMcur2==Ci); % list of nodes in Ci community, without i
Ki_in_i{s} = sum(M(i,CNi,s)); % sum of weights between i and Ci
GQ_i{s} = (K(:,i,s)*SumTot2(:,Ci,s))/(2*(m{s}^(2))) - Ki_in_i{s}/m{s}
    - ((K(:,i,s))^(2))/(2*(m{s}^(2)));
% Recalculate values
SumTot2(:,Ci,s) = SumTot2(:,Ci,s) - K(:,i,s); % weights incident to Ci community
SumIn2(:,Ci,s) = SumIn2(:,Ci,s) - 2*sum(M(i,CNi,s)) - M(i,i,s); % weights community i
end

G = sparse(1,N);
for j=1:length(NB)
    SumTot3=SumTot2;
    SumIn3=SumIn2;
    Cj = COMcur(NB(j)); % community of node j
    if (G(Cj) == 0) % If not tried with another node of community Cj yet
        G(Cj)=1;
        for s=1:k
            % Insert i in the community of j for each layer
            CNj = (COMcur3==Cj); % nodes in community Cj, without j
            Ki_in_j{s} = sum(M(i,CNj,s)); % sum of weights between i and Cj
            GQ_j{s} = Ki_in_j{s}/m{s} - (K(:,i,s)*SumTot3(:,Cj,s))/(2*(m{s}^(2)));
            % gain deltaQ if I put isolated node i in Cj
            % Recalculate
            SumTot3(:,Cj,s) = SumTot3(:,Cj,s) + K(:,i,s);
            SumIn3(:,Cj,s) = SumIn3(:,Cj,s) + 2*sum(M(i,CNj,s)) + M(i,i,s);
        end
        COMcur4=COMcur;
        COMcur4(i)=Cj;
        [COMcur4] = reindex_com(COMcur4);
        COM{1}{3}=COMcur4;
    end
    % variance
    if k==1
        GV_j = 0;
    else
        for s=1:k
            DQ{s} = GQ_i{s} + GQ_j{s}; % gains
        end
        M_DQ = sum([DQ{:}])/k; % gain average
        GV_j = ((sum(((DQ{:}) - M_DQ).^2)) / (k-1)) + (2/(k-1)) * sum(117)
sum(((Q{1}[:]) - Average) .* ([DQ[:]] - M_DQ)); \%gain variance

GF_j = (1-lambda)*M_DQ - lambda*GV_j;
if GF_j > -10^-14
  F_j = Function + GF_j;
  \%modularity
  for s=1:k
    Q_j{s} = Q{1}{s} + DQ{s};
  end
  \%average
  A_j = Average + M_DQ;
  [L] = add_to_filter({Q_j},COM,SumTot3,SumIn3,A_j,F_j,L,k);
end
end
end
end

\%add a point to the filter
function [L] = add_to_filter(QQ,COM,SumTot,SumIn,Average,Function,L,k)
if length(L) ^= 0
  COMcur=COM{1}{1}; \%current community
  \%i. Check if the new point is dominated by a point in the filter
  DD=0;
  l=1;
  while (DD==0) & l<=length(L)
    D=0;
    s=1;
    while (D==0) & s<=k
      if QQ{1}{s}>L{l}{1}{s}
        D=1;
      end
      s=s+1;
    end
    if D %the new point is not dominated by l
      DD=1;
    end
    l=l+1;
  end
  if DD %the new point is not dominated
    DDD=0;
    l=1;
    while (DDD==0) & l<=length(L)
      D=1;
      s=1;
      while D & s<=k

if abs(L{l}{1}{s} - QQ{1}{s}) > 10^(-14)
D=0;
end
s = s + 1;
end
if D
DDD=1;
end
l = l + 1;
end
if DDD==0
%2. Check if the other points in the filter are dominated by the new point
for l=1:length(L)
D=1;
s=1;
while D & s<=k
if L{l}{1}{s}>QQ{1}{s}+10^(-14)
D=0;
end
s=s+1;
end
if D %l is dominated by the new point so I remove it from the filter
L{l}=[];
end
empties = (cellfun(@isempty, L));
L(empties) = [];
L{end+1}=[QQ,COM,SumTot,SumIn,Average,Function]; %I add the new point to the filter
end
end
else
L{end+1}=[QQ,COM,SumTot,SumIn,Average,Function]; %I add the new point to the filter
end
end
% cut the filter
function [L] = cut_filter(L,h,k,lambda)
% calculate the function for every element of the list
if length(L)>h

for l=1:length(L)
    %function
    Function{l} = L{l}{6};
end

%remove the situations in the filter with the lower values
for t=1:(length(L)-h)
    [mn,idx] = min([Function{:}]);
    v=find([Function{:}]== mn); %if more element with same average values, delete the last one (because with h=1 I have same result as cluster_jl_average_variance)
    idx_t=v(end);
    Function{idx_t}=Inf;
    L{idx_t}=[];
end
empties = find(cellfun(@isempty,L));
L(empties) = [];
end
end

%Compute modularity
function MOD = compute_modularity(C,Mat)
S = size(Mat);
N = S(1);
m = sum(sum(Mat))/2; %total weight
MOD = 0;
COMu = unique(C); %list of communities without repetitions and in sorted order
for j=1:length(COMu)
    Cj = (C==COMu(j)); %list of nodes in Cj
    Ec = sum(sum(Mat(Cj,Cj))); %sum of weights between nodes in Cj
    Et = sum(sum(Mat(Cj,:))); %sum of weights of nodes incident in nodes of Cj
    if Et>0
        MOD = MOD + Ec/(2*m)-(Et/(2*m))^2;
    end
end
[C Ss] = reindex_com(COMold);
S = sparse(1,length(COMu));
for l=1:length(COMu)
    S(l) = length(COMold(COMold==COMu(l)));
end
[Ss INDs] = sort(S,'descend');
for l=1:length(COMu)
    C(COMold==COMu(INDs(l))) = l;
end
%Re-index community partition not by size but by initial order
function [C] = reindex_com(COMold)
    C = sparse(1,length(COMold)); % vector with a index for each node
    COMu = unique(COMold); % unique(v)=same data of v but not repetitions
    and in sorted order
    COMolds = sort(COMold);
    COMu = COMolds([true;diff(COMolds(:))>0]);
    for l=1:length(COMu)
        C(COMold==COMu(l)) = l; % Rename the communities
    end
end
Appendix F

Matlab code of the multi-variance-plus method (Section 3.2)

% multi-variance-plus
% Inputs:
% M: vector of weight matrix of each layer
% h: length of the filter
% lambda: weight of variance in function for cut filter
% z: 1 = Recursive computation
% : 0 = Just one level computation
% Output:
% L the filter with the following information
% for each element l:
% l{1}=Q cell with modularity of each layer
% l{2}={COMcur COMfull COMindex} communities in the current graph, in
% the
% original graph, communities in the current graph reindexed
% l{3}=SumTot (sum of weights of the links incident to nodes in a
% community)
% l{4}=SumIn (sum of weights of the links inside a community)
% l{5}=Average (average)
% l{6}=Function (function)
%

function [L ending] = multi_variance_plus(M,lambda,h,z)

if nargin < 1
    error('not enough argument');
end
if nargin < 2
d error('not lambda defined');
end
def nargin < 3
d error('not h defined');
end
def nargin < 4
d z = 1;
end
S = size(M);
N = S(1);
def length(S)==3
d k = S(3);
e else k = 1;
e end
def ending = 0;
def for s=1:k
d M2(:,s)=M(:,s)-diag(diag(M(:,s)));
e end
def total weight of the graph
def for s=1:k
def if z==1
def M(:,s) = M(:,s) + diag(diag(M(:,s)));
e end
def m{=} = sum(sum(M(:,s)))/2;
e end
Niter = 0; %number of iterations
def Construction of the beginning values
def COM{1} = 1:S(1); %current community %At the beginning each node is a community
def COM{2} = 1:S(1); %original graph community
def COM{3} = 1:S(1); %community reindexed
def COM(i)= Community of node i
def for s=1:k
def K(:,s) = sum(M(:,s)); % K(i)= sum of weight incident to node i
def SumTot(:,s) = sum(M(:,s)); %SumTot(i)= sum of weight incident to nodes in community i
def SumIn(:,s) = diag(M(:,s)); %SumIn(i)= sum of weight inside community i (loops)
def Q{s} = compute_modularity(COM{1},M(:,s));
124
Average = sum([Q{:}]) / k;

if k==1
   Variance = 0;
else
   Variance = (sum(([Q{:}]-Average).^2)) / (k-1);
end

% function
Function = (1-lambda) * Average + lambda * Variance;

% filter
L={[Q],{COM},SumTot,SumIn,Average,Function};

% if no edges in any layer or just one node
if (sum(cellfun(@(x)x>0,m)) == 0) | N == 1 % logical or
   ending = 1;
else
   M(:,:,(cellfun(@(x)x==0,m)))=[];
   k = k - sum(cellfun(@(x)x==0,m));

% Neighbor{j}{s} neighbor of node j in layer s
for j=1:N
   for s=1:k
      temp=M2(:,:,:);  
      Neighbor_j{s} = find(temp(j,:));
   end
   Neighbor{j}=Neighbor_j;
end

GAIN = 1;
while (GAIN == 1) % Stop when putting nodes in other communities do not increase the modularity
   % while Niter<=2
   L_old=L;
   L_new=L;
   for i=1:N % for each node, in this order
      L_o=L;
      for l=1:length(L_o) % for each situation in the filter
         % delete the point from the filter
         L_new_o=L_new;
         index = cellfun(@(x) isequal(x,L_o{l}), L_new, 'UniformOutput', 1);
         L_new(index)=[];
         % step1
         [L_new,U] = step_1({L_o{l}{1}},{L_o{l}{2}},{L_o{l}{3}});
      end
   end
end
if no new point, insert again the initial point in the filter
if U==0
    L_new = L_new0;
end
L_new = cut_filter(L_new,h,k,lambda);
L=L_new;
end

% Check if nothing happened
if length(L_old)==length(L_new)
    GAIN=1;
else
    C_old=[];
    C_new=[];
    for l=1:length(L_old)
        R_old=[];
        R_new=[];
        for s=1:k
            R_old=[R_old L_old{l}{1}{s}];
            R_new=[R_new L_new{l}{1}{s}];
        end
        C_old=[C_old; R_old];
        C_new=[C_new; R_new];
    end
    C = ismembertol(C_old,C_new,10^(-6));
    if size(C,1)*size(C,2) == sum(sum(C))
        GAIN=0;
    end
end
Niter = Niter + 1;
tEnd = toc;
if tEnd > 500
    L={};
    return
end

% Perform part 2
if (z==1)
    L = cut_filter(L,1,k); % one final element
    Mnew=M; % new weight matrix of this iteration
    Mold=Mnew; % old weight matrix of iteration before
    COMcur = L{1}{2}{3}; % communities in the graph of this iteration
    COMfull = L{1}{2}{3}; % communities in the original graph
    LL=L;
    j=2; % number of pass (step1+step2)
S = 1;
while S
    L_old = LL;
    Mold = Mnew;
    S2 = size(Mold); % number of nodes
    Nnode = S2(1);
    COMu = unique(COMcur); % list of communities without repetitions
        and in sorted order
    Ncom = length(COMu); % number of communities
    ind_com = sparse(Ncom, Nnode); % zero matrix with a row for each
        community and a column for each node in this configuration
    ind_com_full = sparse(Ncom, N); % zero matrix with a row for each
        community and a column for each node of the original graph
    for p = 1 : Ncom % for each community
        ind = find(COMcur == p);
        ind_com(p, 1 : length(ind)) = ind; % in row p insert the indeces
        of the nodes in community p in this configuration
    end
    for p = 1 : Ncom
        ind = find(COMfull == p);
        ind_com_full(p, 1 : length(ind)) = ind; % in row p insert the
            indeces of the nodes in community p in the original graph
    end
    Mnew = [];
    for s = 1 : k
        Mnew(:, :, s) = sparse(Ncom, Ncom); % new matrix (each node is
            a community)
        for m = 1 : Ncom
            for n = m : Ncom
                ind1 = ind_com(m, :);
                ind2 = ind_com(n, :);
                % weights of edges between communities
                Mnew(m, n, s) = sum(sum(Mold(ind1(ind1 > 0), ind2(ind2 > 0)
                    , s)));
                Mnew(n, m, s) = sum(sum(Mold(ind1(ind1 > 0), ind2(ind2 > 0)
                    , s)));
            end
        end
    % apply first step to this new matrix but z=0 without recursive
    LLL e = multi_variance_plus(Mnew, lambda, h, 0);
    if isempty(LLL)
        L = LLL;
        return
    else
        LL = cut_filter(LLL, 1, k); % one final element
        COMfull = sparse(1, N); % communities of the original graph
    end
end
end
end
end
COMcur = LL{1}{2}{3}; % communities
for p = 1:Ncom
    ind1 = ind_com_full(p,:); % nodes in community p in the
    % original graph
    COMfull(ind1(ind1>0)) = COMcur(p); % community now of
    % node i
end
[COMfull] = reindex_com(COMfull);
LL{1}{2}{2} = COMfull;
% communities do not change
if isequal(L_old{1}{2}{2}, LL{1}{2}{2})
    % return not just this value, but all the value of the
    % last list
    % calculate COMfull for each element of the last list
    for l = 1:length(LLL)
        COMfull = sparse(1,N); % communities of the original
        % graph
        COMcur = LLL{l}{2}{3}; % communities
        for p = 1:Ncom
            ind1 = ind_com_full(p,:); % nodes in community p
            % in the original graph
            COMfull(ind1(ind1>0)) = COMcur(p); % community
            % now of node i
        end
        [COMfull] = reindex_com(COMfull);
        LLL{l}{2}{2} = COMfull;
    end
    L = LLL;
    S = 0;
    j = j + 1; % start another pass
    tEnd = toc;
    if tEnd > 600
        L = {};
        return
    end
    % Compute step_1
    function [L,U] = step_1(Q,COM,SumTot,SumIn,Average,Function,k,Neighbor,K
    ,M,N,m,L,h,lambda,i)
    U = 0; % if U=1 add at least an element to the list, otherwise insert again
    % the initial point in the filter
    COMcur = COM{1}{1}; % current community
    Ci = COMcur(i); % community of node i
    % Compute step_1
    function [L,U] = step_1(Q,COM,SumTot,SumIn,Average,Function,k,Neighbor,K
    ,M,N,m,L,h,lambda,i)
NB = unique (cat(2, Neighbor{i}{:}));
SumTot2 = SumTot;
SumIn2 = SumIn;

% remove i from its community
for s=1:k
    COMcur2 = COMcur;
    COMcur2(i) = -1;
    CNi = (COMcur2==Ci);  % list of nodes in Ci community, without i
    Ki_in_i{s} = sum(M(i,CNi,s));  % sum of weights between i and Ci
    GQ_i{s} = (K(:,i,s)*SumTot2(:,Ci,s))/(2*(m{s}^(2))) - Ki_in_i{s}/m{s} - ((K(:,i,s))^(2))/(2*(m{s}^(2)));
end

% Recalculate values
SumTot2(:,Ci,s) = SumTot2(:,Ci,s) - K(:,i,s);  % weights incident to Ci community
SumIn2(:,Ci,s) = SumIn2(:,Ci,s) - 2*sum(M(i,CNi,s)) - M(i,i,s);  % weights community i

G = sparse(1,N);
for j=1:length(NB)
    SumTot3 = SumTot2;
    SumIn3 = SumIn2;
    Cj = COMcur(NB(j));  % community of node j
    if (G(Cj) == 0)  % If have not tried with another node of community Cj yet
        G(Cj) = 1;
        for s=1:k
            COMcur3 = COMcur;
            COMcur3(i) = -1;
            CNj = (COMcur3==Cj);  % nodes in community Cj, without j
            Ki_in_j{s} = sum(M(i,CNj,s));  % sum of weights between i and Cj
            GQ_j{s} = Ki_in_j{s}/m{s} - (K(:,i,s)*SumTot3(:,Cj,s))/(2*(m{s}^(2)));
        end
        COMcur4 = COMcur;
        COMcur4(i) = Cj;
        COM{1}{1} = COMcur4;
        [COMcur4] = reindex_com(COMcur4);
        COM{1}{3} = COMcur4;
    end
end
% variance
if k == 1
    GV_j = 0;
else
    for s = 1:k
        DQ{s} = GQ_i{s} + GQ_j{s}; % gains
    end
    M_DQ = sum([DQ{:}])/k; % gain average
    GV_j = ((sum(([DQ{:}]-M_DQ).^2)) / (k-1)) + (2/(k-1)) * sum(
        sum(([Q{1}{:}] - Average) .* ([DQ{:}] - M_DQ))); % gain variance
end
GF_j = (1 - lambda)*M_DQ + lambda*GV_j;
if GF_j > -10^(-14)
    F_j = Function + GF_j;
    % modularity
    for s = 1:k
        Q_j{s} = Q{1}{s} + DQ{s};
    end
    % average
    A_j = Average + M_DQ;
    [L] = add_to_filter({Q_j},COM,SumTot3,SumIn3,A_j,F_j,L,k);
    U = 1;
end
end
end
end
% add a point to the filter
function [L] = add_to_filter(QQ,COM,SumTot,SumIn,Average,Function,L,k)
if length(L) ~= 0
    COMcur = COM{1}{1}; % current community
else
    DD = 0;
    l = 1;
    while (DD == 0) & l <= length(L)
        D = 0;
        s = 1;
        while (D == 0) & s <= k
            if QQ{1}{s} > L{l}{1}{s}
                D = 1;
            end
            s = s + 1;
        end
        if D % the new point is not dominated by l
            DD = 1;
        end
    end
end
end
l=l+1;

if DD %the new point is not dominated
    DDD=0;
l=1;
    while (DDD==0) & l<=length(L)
        D=1;
s=1;
        while D & s<=k
            if abs(L{l}{1}{s} - QQ{1}{s}) > 10^{-14}
                D=0;
            end
            s = s + 1;
        end
        if D
            DDD=1;
        end
        l = l + 1;
    end
if DDD==0

%2. Check if the other points in the filter are dominated by the new point %
for l=1:length(L)
    D=1;
s=1;
    while D & s<=k
        if L{l}{1}{s}>QQ{1}{s}+10^{-14}
            D=0;
        end
        s=s+1;
    end
    if D %l is dominated by the new point so I remove it from the filter
        L{l}=[];
    end
end
empties =(cellfun (@isempty,L));
L(empties) = [];
L{end+1}=[QQ,COM,SumTot,SumIn,Average,Function]; %I add the new point to the filter

else

end
end
end

131
L{end+1} = [QQ, COM, SumTot, SumIn, Average, Function];  % I add the new point to the filter
end

% cut the filter
function [L] = cut_filter(L, h, k, lambda)
% calculate the function for every element of the list
if length(L) > h
    for l = 1:length(L)
        % function
        Function{l} = L{l}{6};
    end

% remove the situations in the filter with the lower values
for t = 1:(length(L)-h)
    [mn, idx] = min([Function{:}]);
    v = find([Function{:}] == mn);  % if more element with same average values,
    idx_t = v(end);  % delete the last one (because with h=1 I have same result as
    Function{idx_t} = Inf;
    L{idx_t} = [];
end
empty = find(cellfun(@isempty, L));
L(empty) = [];
end

% Compute modularity
function MOD = compute_modularity(C, Mat)
S = size(Mat);
N = S(1);
m = sum(sum(Mat))/2;  % total weight
MOD = 0;
COMu = unique(C);  % list of communities without repetitions and in
sorted order
for j = 1:length(COMu)
    Cj = (C == COMu(j));  % list of nodes in Cj
    Ec = sum(sum(Mat(Cj, Cj)));  % sum of weights between nodes in Cj
    Et = sum(sum(Mat(Cj,:)));  % sum of weights of nodes incident in nodes of Cj
    if Et > 0
MOD = MOD + Ec/(2*m) - (Et/(2*m))^2;
end
end

% Re-index community partition by size
function [C Ss] = reindex_com(COMold)
C = sparse(1,length(COMold));
COMu = unique(COMold);
S = sparse(1,length(COMu));
for l=1:length(COMu)
    S(l) = length(COMold(COMold == COMu(l)));
end
[Ss INDs] = sort(S,'descend');
for l=1:length(COMu)
    C(COMold == COMu(INDs(l))) = l;
end
end

% Re-index community partition not by size but by initial order
function [C] = reindex_com(COMold)
C = sparse(1,length(COMold));
COMu = unique(COMold);
COMolds = sort(COMold);
COMu = COMolds([true;diff(COMolds(:))>0]);
for l=1:length(COMu)
    C(COMold == COMu(l)) = 1;
end
Bibliography


138


textual data with relations,” in Proceedings of the 17th ACM conference on Information

[75] G.-J. Qi, C. C. Aggarwal, and T. Huang, “Community detection with edge content in
social media networks,” in 2012 IEEE 28th International Conference on Data Engineering,

[76] Y. Zhou, H. Cheng, and J. X. Yu, “Graph clustering based on structural/attribute similar-

graph clustering,” in Proceedings of the 2012 ACM SIGMOD international conference on

[78] A. Silva, W. Meira Jr, and M. J. Zaki, “Mining attribute-structure correlated patterns in

using content and links,” in Proceedings of the 22nd international conference on World


graphs: A spectral perspective,” IEEE Transactions on Signal Processing, vol. 60, no. 11,

large dense graph databases,” in Proceedings of the 12th ACM SIGKDD international con-
ference on Knowledge discovery and data mining, pp. 797–802, 2006.

eleventh ACM SIGKDD international conference on Knowledge discovery in data mining,

2020.


[86] P.-Y. Chen and A. O. Hero, “Multilayer spectral graph clustering via convex layer aggrega-
tion: Theory and algorithms,” IEEE Transactions on Signal and Information Processing


