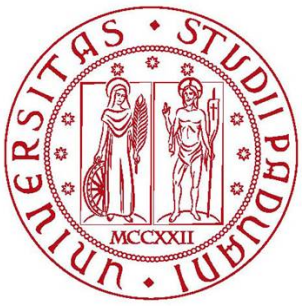


University of Padua

Department of Mathematics "Tullio Levi Civita"

Master Degree in Mathematics

Methods for community detection in multi-layer networks

Surpervisor: Prof. Francesco Rinaldi
University of Padua

Co-surpervisor: 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

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 $(N_s,E_s)_{s=1,\dots,k}$ 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 $(A_s)_{s=1,\dots,k}$ 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

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - P_{ij}) \delta(C_i, C_j) \tag{1.1}$$

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 δ 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

$$Q = \frac{1}{2m} \sum_{i,j} \Big(A_{ij} - \frac{k_i k_j}{2m} \Big) \delta(C_i, C_j) \tag{1.2}$$

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 ΔF is the decrease of the function and β is an index of stochastic noise, which increases after each iteration. Simulated annealing was first used for modularity optimization by *Guimerá 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 quinckly 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 $\Delta Q1_i$ 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 $\Delta Q2_{i\to j}$ on each layer. Now the algorithm calculates the gain $\Delta F_{i\to j}$ 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 $\Delta F_{i\to j^*}$ 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 literaturate, 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:
3:     repeat
4:         PHASE 1
5:         Initialize:
6:             C ← initial partition, where each vertex of graph G is a community
7:             Q ← modularity vector of the initial partition
8:             F ← function value of the initial partition
9:             NB ← neighbour nodes vector
10:        repeat
11:
12:            for each node i do
13:                remove node i from its community
14:                ΔQ1_i ← modularity gain for removing node i from its community
15:
16:                for each node j that is neighbour of node i do
17:                    insert node i into community of node j
18:                    ΔQ2_{i→j} ← modularity gain for inserting i into community of j
19:                    ΔF_{i→j} ← function gain for changing the community of node i
20:                end for
21:                ΔF_{i→j*} ← best function gain
22:
23:                if ΔF_{i→j*} >0 then
24:                    move node i into community of node j*
25:                    Q ← modularity vector of the new partition
26:                    F ← function value of the new partition
27:                end if
28:            end for
29:
30:        until no improved clustering found.
31:
32:        PHASE 2
33:        G ← reduced graph where each community of partition C is a node
34:
35:    until no improved clustering found.
36:    return C
37: end function
```

therefore we show also the corresponding formula for weighted graphs

$$Q_{\mathrm{w}} = \frac{1}{2w} \sum_{i,j} \Big(W_{ij} - \frac{s_i s_j}{2w} \Big) \delta(C_i, C_j) \tag{3.1}$$

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 δ 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),

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

where C=(C_1,...,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 $\Delta Q1_i$ obtained by moving a node i from its community c can easily be computed by

$$\Delta Q1_i = \frac{\sum_{tot} \cdot k_i}{2m^2} - \frac{{k_i}^2}{2m^2} - \frac{k_{i,in}}{m} \tag{3.3}$$

where $\sum_{tot}$ is the sum of weights of the links incident to node in c, k_i is the degree of node i, $k_{i,in}$ 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 $\Delta Q2_j$ when an isolated node i is moved into the community c of node j

$$\Delta Q2_{i \to j} = \frac{\sum_{i,in}}{m} - \frac{\sum_{tot} \cdot k_i}{2m^2} \tag{3.4}$$

where $\sum_{i,in}$ is the sum of weights of the links inside community c, $\sum_{tot}$ 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

$$M_Q = \frac{\sum_{s=1}^{k} Q_s}{k} \tag{3.5}$$

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 follow

$$\Delta M_Q = \frac{\sum_{s=1}^{k} \Delta Q_s}{k} \tag{3.6}$$

where k is the number of layers and ΔQ_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

$$V_Q = \frac{\sum_{s=1}^{k} (Q_s - M_Q)^2}{k-1} \tag{3.7}$$

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 function

$$F_{\text{-}} = (1-\gamma) M_Q - \gamma V_Q \tag{3.8}$$

$$F_{+} = (1-\gamma) M_Q + \gamma V_Q \tag{3.9}$$

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

$$\Delta F_{\text{-}} = (1-\gamma)\Delta M_Q - \gamma\Big(V_{\Delta Q} + \frac{2}{k-1}(Q - M_Q)^t(\Delta Q - \Delta M_Q)\Big) \tag{3.10}$$

$$\Delta F_{+} = (1-\gamma)\Delta M_Q + \gamma\Big(V_{\Delta Q} + \frac{2}{k-1}(Q - M_Q)^t(\Delta Q - \Delta M_Q)\Big) \tag{3.11}$$

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

$$V_{\Delta Q} = \frac{\sum_{s=1}^{k} (\Delta Q_s - \Delta M_Q)^2}{k-1} \tag{3.12}$$

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*

$$\begin{aligned} z_i^1 &\geq z_i^2 \quad \textit{for each index i=1,..,k and} \\ z_j^1 &> z_j^2 \quad \textit{for at least one index j=1,..,k.} \end{aligned}$$

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,..,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 $\Delta Q1_i$ 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 $\Delta Q2_{i\to j}$ on each layer. Now the algorithm calculates the gain $\Delta F_{i\to j}$ 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 delates 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.

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
$\Delta Q1_i \leftarrow$ 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
$\Delta Q2_{i\to j} \leftarrow$ modularity gain for inserting i into community of j
$\Delta F_{i\to j} \leftarrow$ function gain for changing the community of node i

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

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

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 reffered 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

$$NMI(C^*, C) = \frac{2 \cdot I(C^*, C)}{H(C^*) + H(C)} \tag{4.1}$$

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

$$I(C^*, C) = H(C^*) - H(C^*|C) \tag{4.2}$$

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

$$NMI(C^*, C) = \frac{-2 \cdot \sum_{i=1}^{|C|} \sum_{i=1}^{|C^*|} T_{ij} \cdot log\Big(\frac{T_{ij}t}{T_{i.}\,T_{.j}}\Big)}{\sum_{i=1}^{|C|} T_{i.} \cdot log\Big(\frac{T_{i.}}{t}\Big) + \sum_{j=1}^{|C^*|} T_{.j} \cdot log\Big(\frac{T_{.j}}{t}\Big)} \tag{4.3}$$

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_{in} and p_{out}. These parameters represent the edge probabilities: given nodes v_i and v_j the probability of observing an edge between them is p_{in} (resp. p_{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 $p_{in} >> p_{out}$ on the informative layers and $p_{in} = p_{out}$ on the noisy layers.
In particular, we created networks with 4 communities of 125 nodes each and with $k = 2$, 3 layers, by fixing $p_{in} = 0.1$ and varying p_{out}. In the noisy layers, we fixed $p_{in} = p_{out} = 0.1$.

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_- in equation (3.8) and F_+ in equation (3.9), we set $\gamma = 0.1$, 0.3, 0.5. The ratio between p_{in} and p_{out} changes between columns and it is specified in the second row. For each method and ratio between p_{in} and p_{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_{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.
The results get worse as the ratio between p_{in} and p_{out} decreases, in fact intuitively the communities are less defined when p_{out} is close to p_{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 $p_{in}/p_{out} = 3$, 2.5, a little less well in the third case with $p_{in}/p_{out} = 2$ and get worst in the last situation with $p_{in}/p_{out} = 1.5$.

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

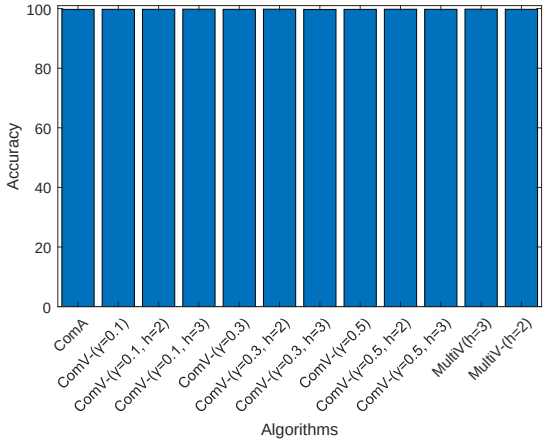
In the noisy case just one layer is informative, so it has a community structure, and all the other layers are just noise, so they give wrong information about the clustering.

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.

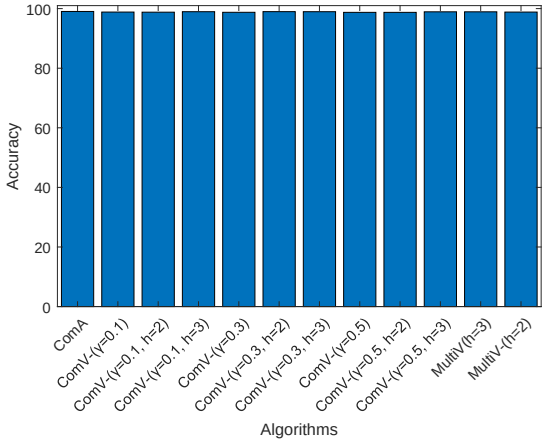
Table 4.5 shows the average results for the noisy case on graphs with $k = 2$ layers. Figure 4.17 and Figure 4.18 represent the Accuracy and the NMI of the results in some bar plots. We

INFORMATIVE CASE: k=2												
	p_{in}/p_{out}=3			p_{in}/p_{out}=2.5			p_{in}/p_{out}=2			p_{in}/p_{out}=1.5		
	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu
ComA	99.74	0.989	2.52	**99.04**	**0.962**	3.98	75.04	0.575	6.96	21.84	0.045	5.36
MultiA h=2	99.76	0.990	140.72	98.84	0.955	214.42	74.22	0.577	254.66	21.08	0.049	287.80
MultiA h=3	**99.80**	**0.992**	147.78	98.92	0.957	328.00	78.20	0.620	425.17	21.58	0.047	451.14
ComV- γ=0.1	99.76	0.990	2.86	98.84	0.955	5.05	75.44	0.588	8.03	21.02	0.043	7.14
ComV- γ=0.3	99.76	0.990	3.46	98.76	0.952	5.35	73.86	0.577	7.77	21.00	0.041	6.90
ComV- γ=0.5	99.74	0.990	6.05	98.74	0.952	5.66	74.74	0.591	7.41	21.46	0.042	7.51
MultiV- γ=0.1, h=2	99.76	0.990	163.26	98.80	0.954	248.27	77.08	0.609	262.40	21.44	0.050	292.21
MultiV- γ=0.1, h=3	**99.80**	**0.992**	170.17	98.96	0.959	446.78	75.72	0.588	421.59	**22.08**	**0.051**	437.92
MultiV- γ=0.3, h=2	**99.80**	**0.992**	169.90	98.98	0.960	250.46	**81.64**	**0.656**	262.39	21.10	0.045	295.12
MultiV- γ=0.3, h=3	99.72	0.988	197.79	98.96	0.959	311.20	76.70	0.593	435.59	20.90	0.043	452.44
MultiV- γ=0.5, h=2	99.78	0.991	106.79	98.74	0.951	228.23	80.72	0.642	266.18	21.40	**0.051**	287.89
MultiV- γ=0.5, h=3	99.76	0.990	185.08	98.92	0.958	331.87	76.84	0.579	419.15	21.70	0.046	460.25

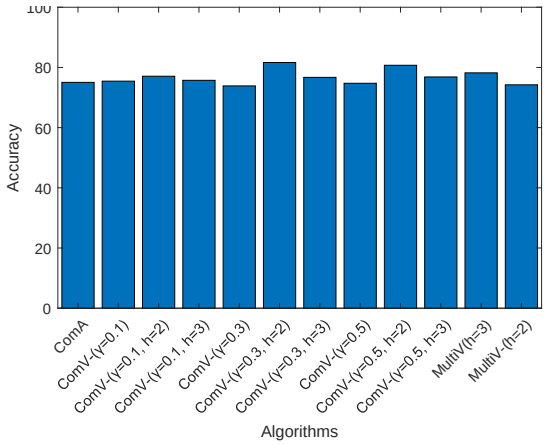
Table 4.1: Experiments in the informative case on artificial graphs with k=2 layers. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.



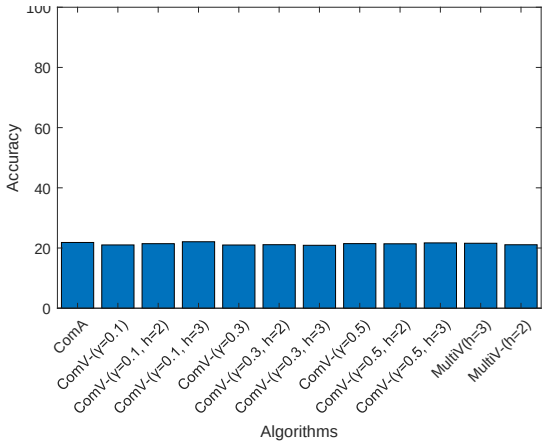
(a) p_{in}/p_{out}=3



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

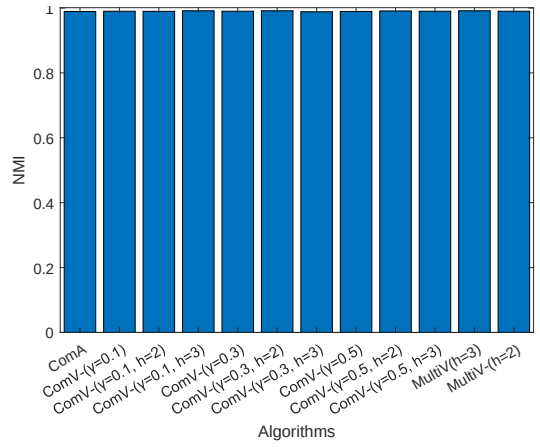


(c) p_{in}/p_{out}=2

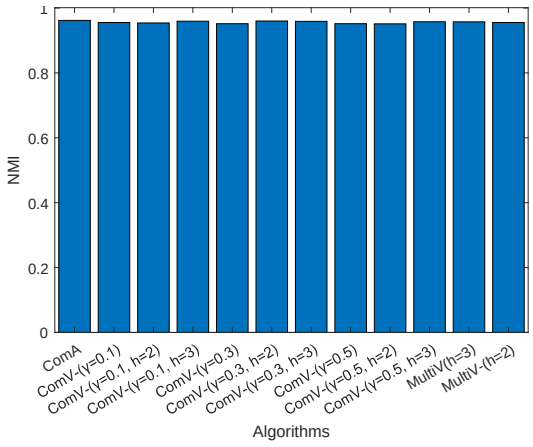


(d) p_{in}/p_{out}=1.5

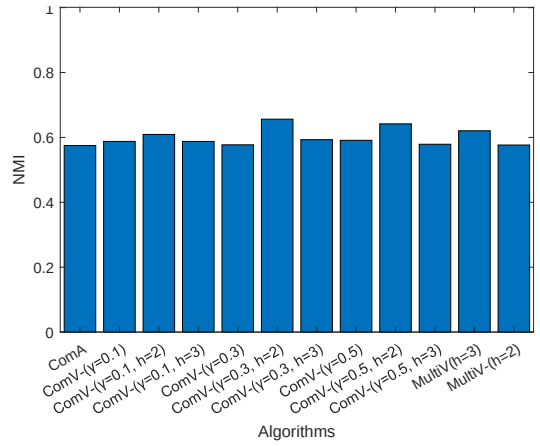
Figure 4.1: Accuracy of the experiments in the informative case on artificial graphs with k=2 layers



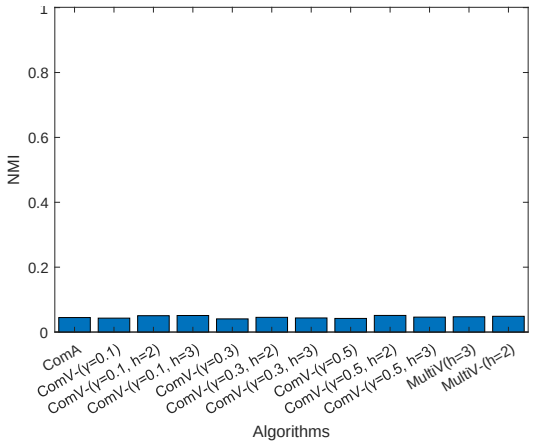
(a) p_{in}/p_{out}=3



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

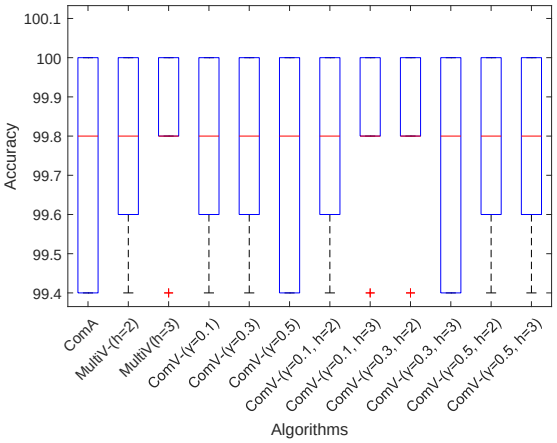


(c) p_{in}/p_{out}=2

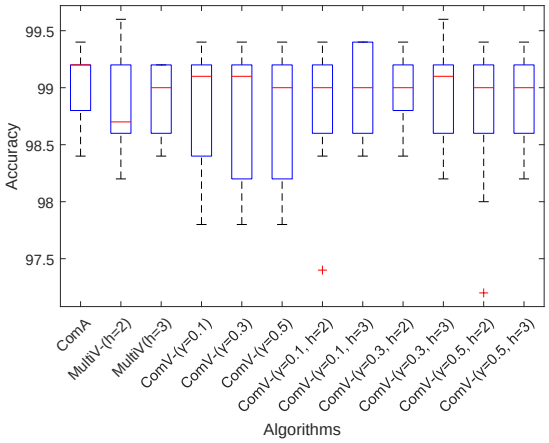


(d) p_{in}/p_{out}=1.5

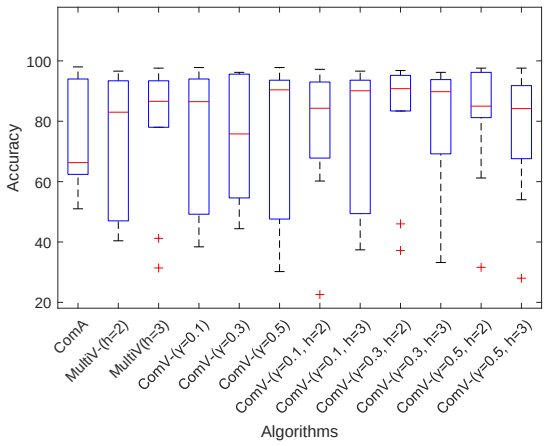
Figure 4.2: NMI of the experiments in the informative case on artificial graphs with k=2 layers



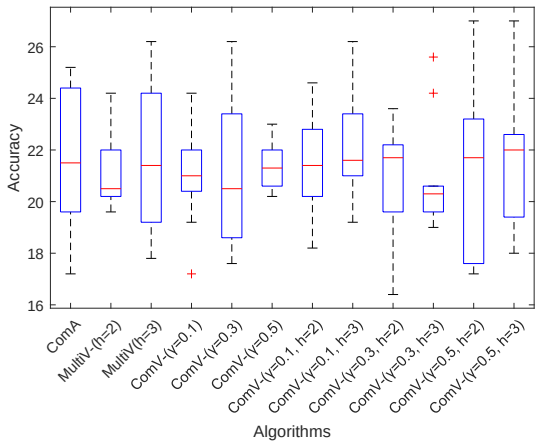
(a) p_{in}/p_{out}=3



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

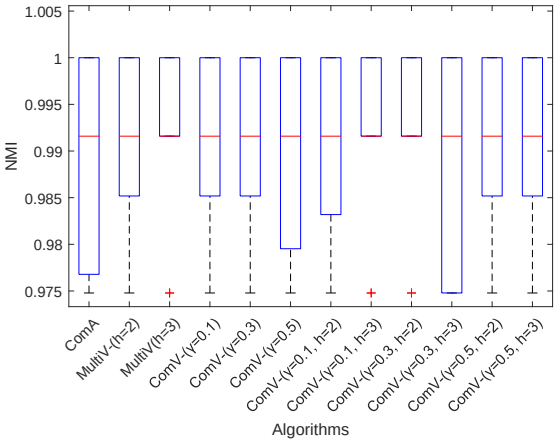


(c) p_{in}/p_{out}=2

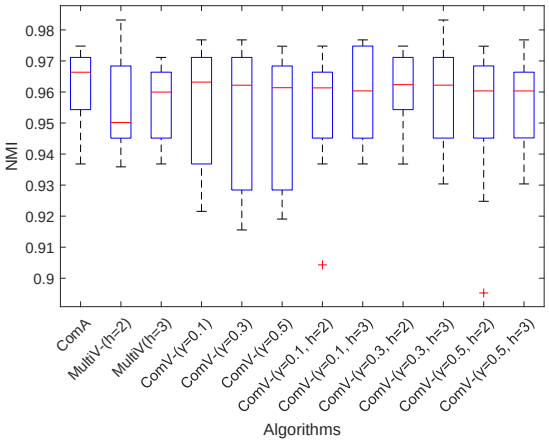


(d) p_{in}/p_{out}=1.5

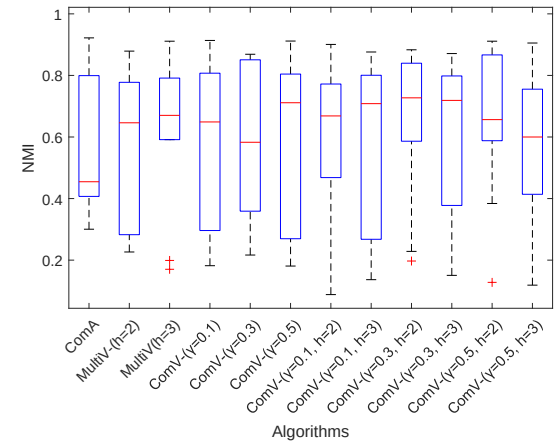
Figure 4.3: Accuracy of the experiments in the informative case on artificial graphs with k=2 layers



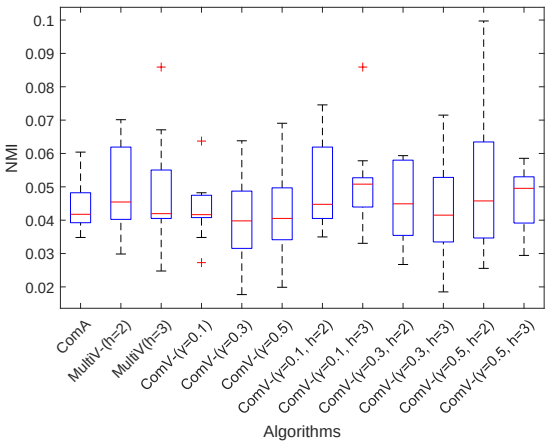
(a) p_{in}/p_{out}=3



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



(c) p_{in}/p_{out}=2

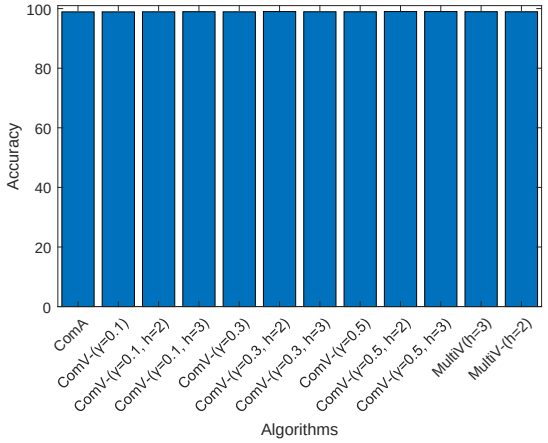


(d) p_{in}/p_{out}=1.5

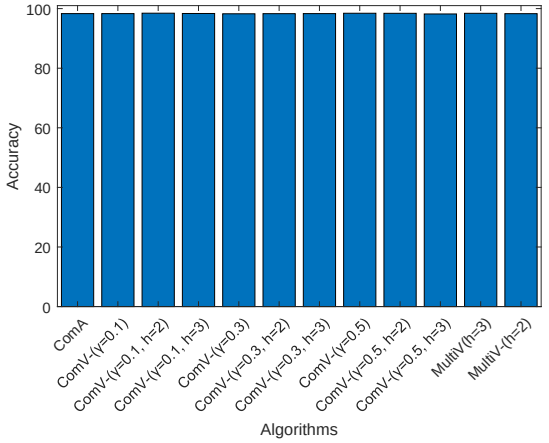
Figure 4.4: NMI of the experiments in the informative case on artificial graphs with k=2 layers

INFORMATIVE CASE: k=2												
	p_{in}/p_{out}=2.6			p_{in}/p_{out}=2.4			p_{in}/p_{out}=2.2			p_{in}/p_{out}=2		
	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu
ComA	98.90	0.956	3.13	98.32	0.936	3.99	93.28	0.833	4.15	75.04	0.575	6.96
MultiA h=2	98.94	**0.975**	139.63	98.30	0.934	197.19	96.78	0.888	219.62	74.22	0.577	254.66
MultiA h=3	98.96	0.958	227.07	98.42	0.939	276.52	97.16	0.895	295.49	78.20	0.620	425.17
ComV- γ=0.1	98.90	0.956	4.20	98.32	0.935	4.90	**97.26**	**0.898**	4.98	75.44	0.588	8.03
ComV- γ=0.3	98.92	0.956	4.29	98.26	0.933	4.92	97.00	0.891	5.06	73.86	0.577	7.77
ComV- γ=0.5	98.94	0.957	4.59	98.44	0.940	4.65	97.20	0.896	4.83	74.74	0.591	7.41
MultiV- γ=0.1, h=2	98.92	0.956	144.55	**98.46**	**0.941**	200.38	96.70	0.880	203.30	77.08	0.609	262.40
MultiV- γ=0.1, h=3	98.96	0.958	216.71	98.36	0.937	286.79	97.06	0.890	296.29	75.72	0.588	421.59
MultiV- γ=0.3, h=2	**99.00**	0.959	180.89	98.30	0.935	192.75	96.52	0.885	200.45	**81.64**	**0.656**	262.39
MultiV- γ=0.3, h=3	98.94	0.957	240.51	98.32	0.936	287.64	95.86	0.874	295.01	76.70	0.593	435.59
MultiV- γ=0.5, h=2	**99.00**	0.959	144.67	98.42	0.939	188.69	96.98	0.888	199.50	80.72	0.642	266.18
MultiV- γ=0.5, h=3	**99.00**	0.959	231.16	98.20	0.931	280.32	96.82	0.886	297.55	76.84	0.579	419.15

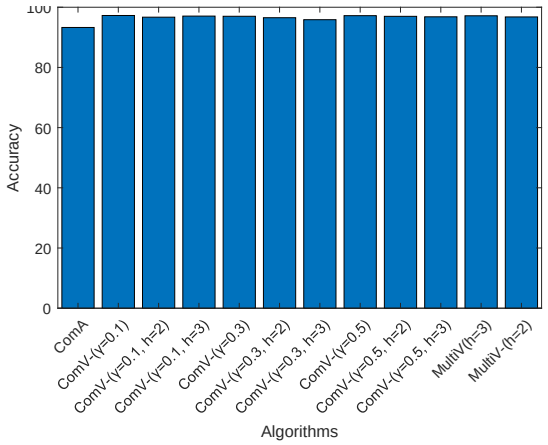
Table 4.2: Experiments in the informative case on artificial graphs with k=2 layers. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.



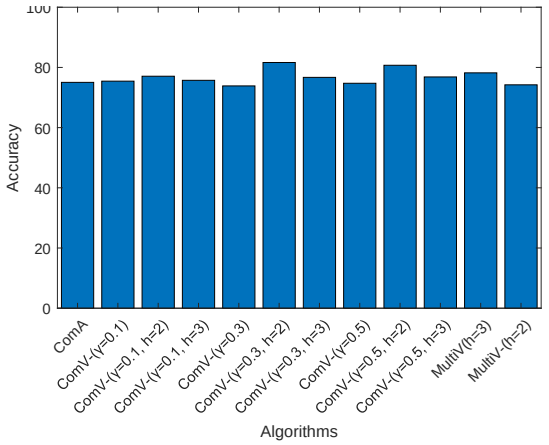
(a) p_{in}/p_{out}=2.6



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

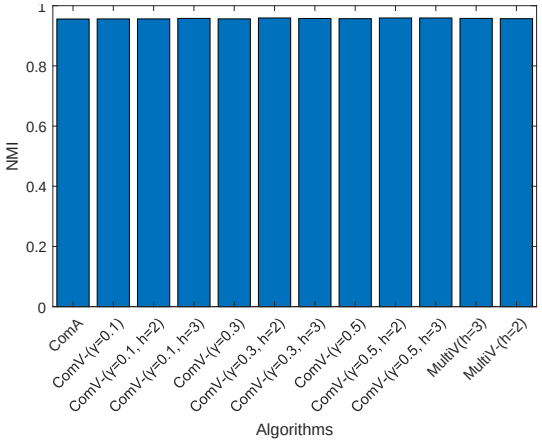


(c) p_{in}/p_{out}=2.2

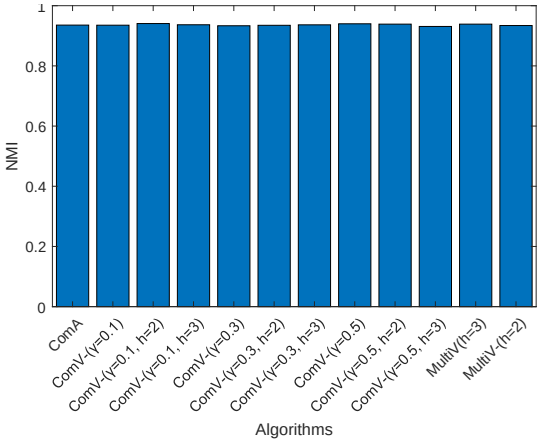


(d) p_{in}/p_{out}=2

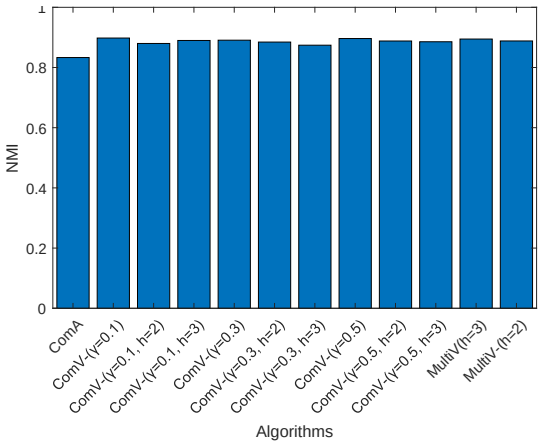
Figure 4.5: Accuracy of the experiments in the informative case on artificial graphs with k=2 layers



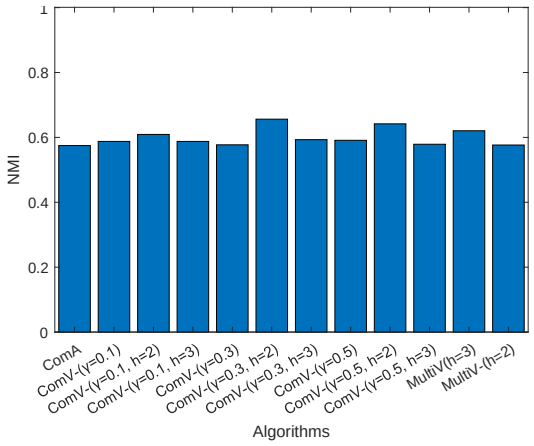
(a) p_{in}/p_{out}=2.6



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

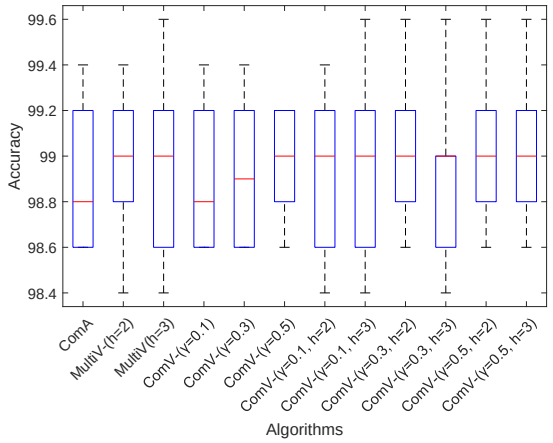


(c) p_{in}/p_{out}=2.2

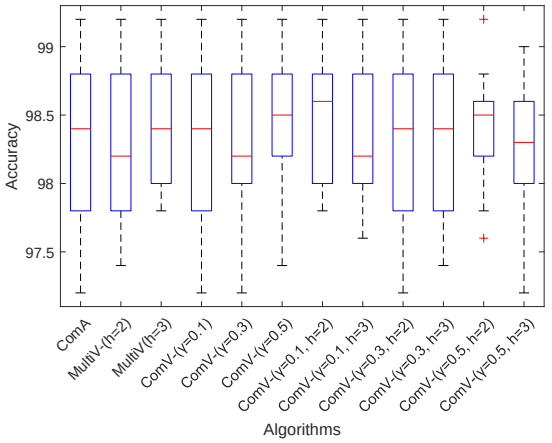


(d) p_{in}/p_{out}=2

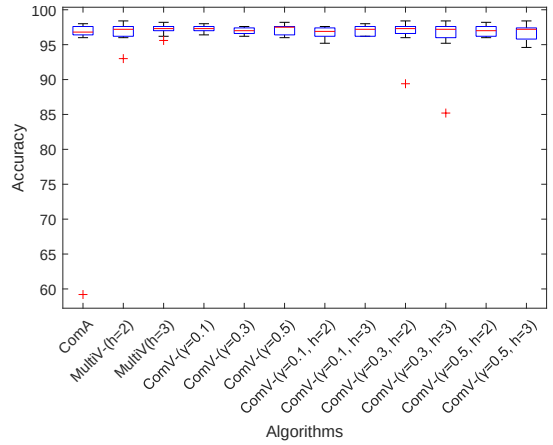
Figure 4.6: NMI of the experiments in the informative case on artificial graphs with k=2 layers



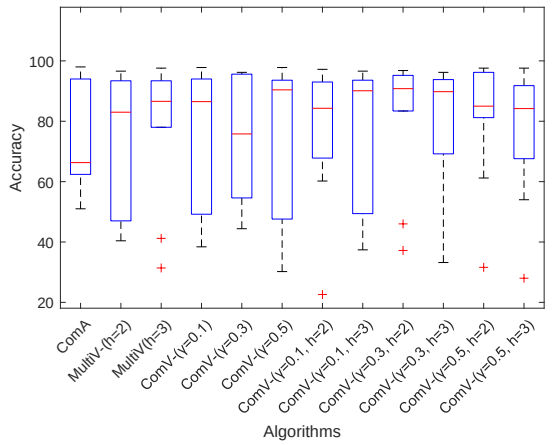
(a) p_{in}/p_{out}=2.6



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

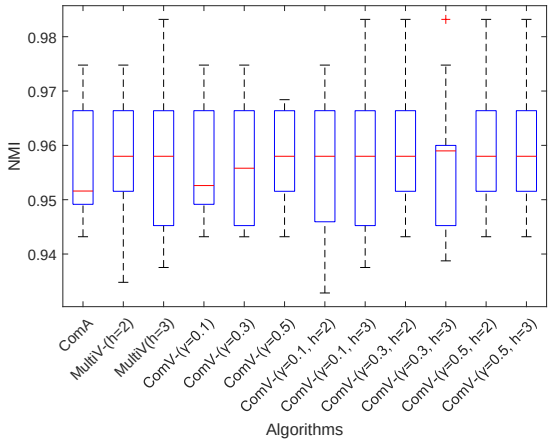


(c) p_{in}/p_{out}=2.2

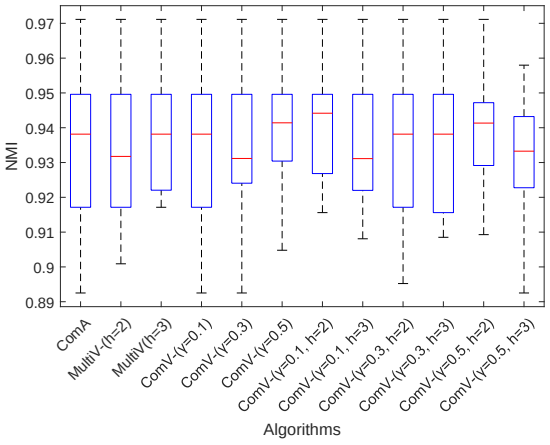


(d) p_{in}/p_{out}=2

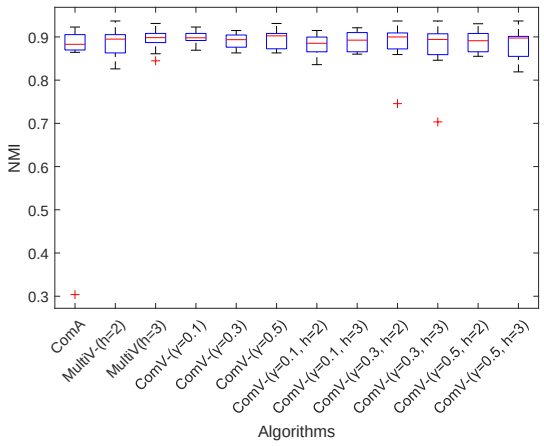
Figure 4.7: Accuracy of the experiments in the informative case on artificial graphs with k=2 layers



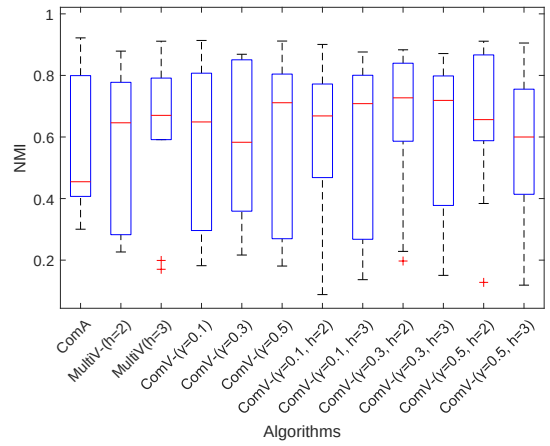
(a) p_{in}/p_{out}=2.6



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



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

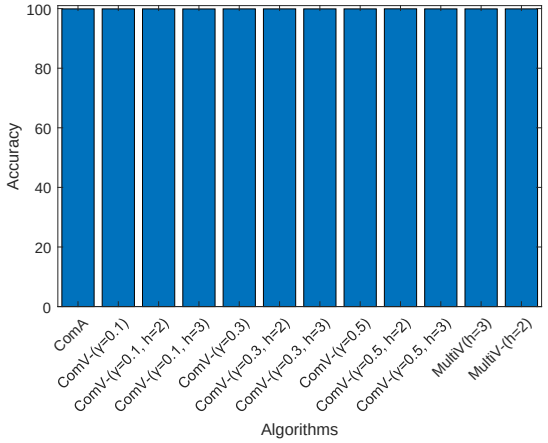


(d) p_{in}/p_{out}=2

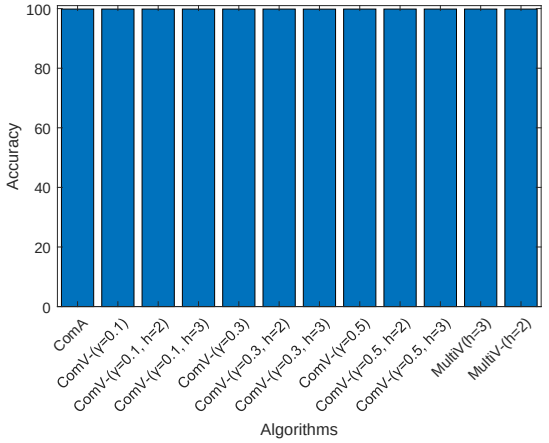
Figure 4.8: NMI of the experiments in the informative case on artificial graphs with k=2 layers

INFORMATIVE CASE: k=3												
	p_{in}/p_{out}=3			p_{in}/p_{out}=2.5			p_{in}/p_{out}=2			p_{in}/p_{out}=1.5		
	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu
ComA	99.90	0.996	5.11	**99.84**	**0.993**	5.42	98.80	0.952	5.42	25.56	0.076	7.20
MultiA h=2	99.90	0.996	231.03	**99.84**	**0.993**	264.99	98.80	0.952	275.88	24.38	0.079	325.63
MultiA h=3	99.90	0.996	358.93	**99.84**	**0.993**	408.75	98.80	0.952	417.78	26.20	0.076	477.31
ComV- γ=0.1	99.90	0.996	5.94	**99.84**	**0.993**	6.19	98.72	0.949	6.31	25.36	0.072	10.13
ComV- γ=0.3	99.90	0.996	6.09	**99.84**	**0.993**	6.11	98.74	0.950	6.36	27.48	0.093	9.75
ComV- γ=0.5	99.90	0.996	5.67	**99.84**	**0.993**	5.75	98.72	0.949	6.29	**29.00**	0.096	8.68
MultiV- γ=0.1, h=2	99.90	0.996	231.36	**99.84**	**0.993**	271.00	98.70	0.948	277.56	24.24	0.071	312.35
MultiV- γ=0.1, h=3	99.88	0.995	352.77	**99.84**	**0.993**	402.17	98.84	0.953	426.00	27.88	0.092	507.82
MultiV- γ=0.3, h=2	99.90	0.996	273.54	**99.84**	**0.993**	266.89	**98.86**	**0.954**	231.57	25.00	0.076	334.74
MultiV- γ=0.3, h=3	99.90	0.996	354.30	99.82	0.992	406.74	98.78	0.952	425.64	24.44	0.072	509.90
MultiV- γ=0.5, h=2	**99.92**	**0.997**	234.06	**99.84**	**0.993**	268.40	98.80	0.952	277.64	24.76	0.072	337.01
MultiV- γ=0.5, h=3	99.90	0.996	346.97	**99.84**	**0.993**	396.08	98.82	0.953	422.18	26.98	**0.104**	521.42

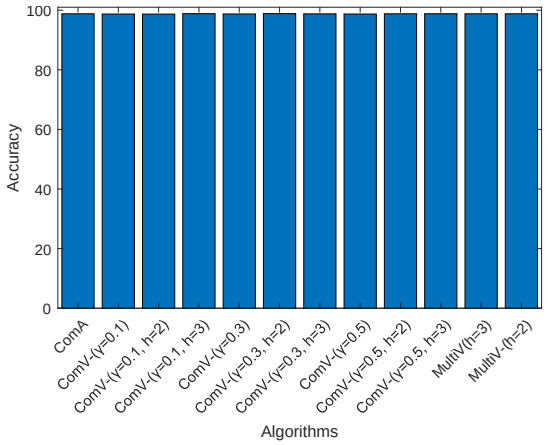
Table 4.3: Experiments in the informative case on artificial graphs with k=3 layers. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.



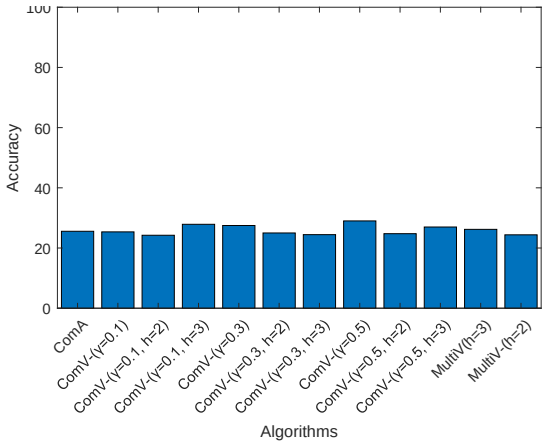
(a) p_{in}/p_{out}=3



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

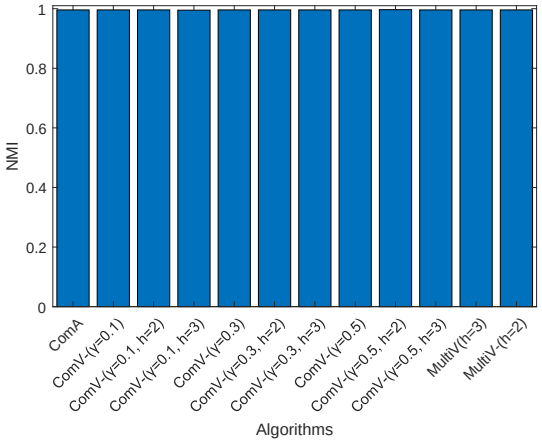


(c) p_{in}/p_{out}=2

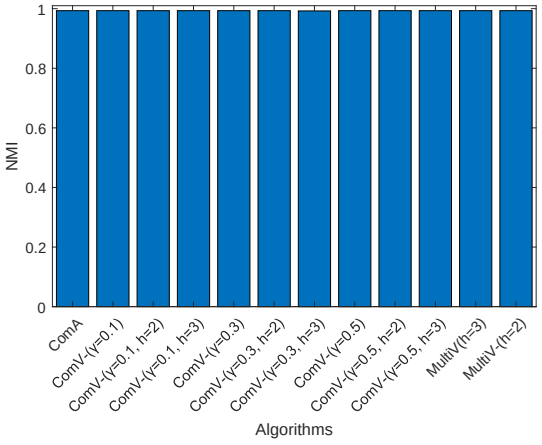


(d) p_{in}/p_{out}=1.5

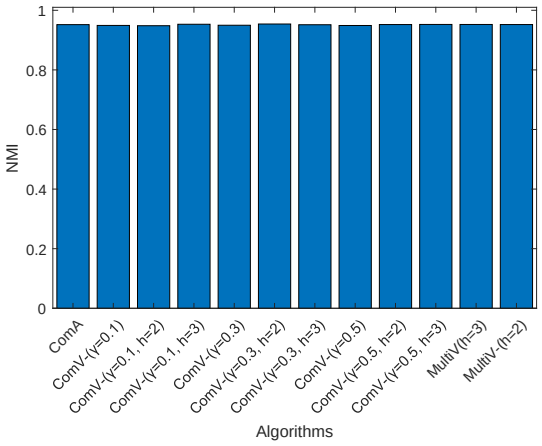
Figure 4.9: Accuracy of the experiments in the informative case on artificial graphs with k=3 layers



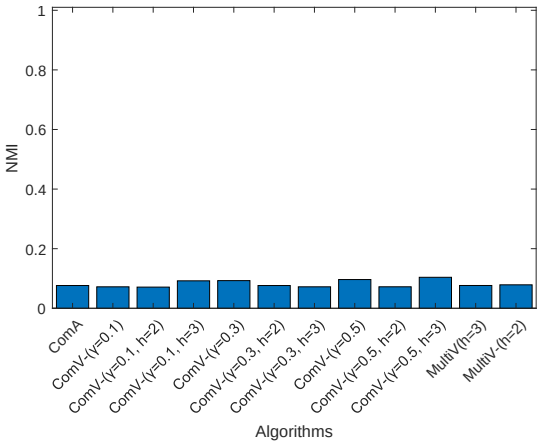
(a) p_{in}/p_{out}=3



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

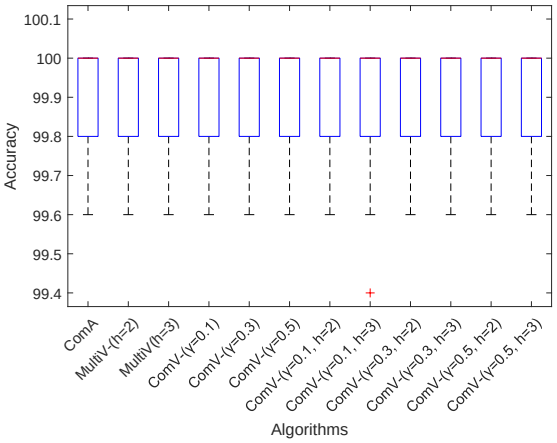


(c) p_{in}/p_{out}=2

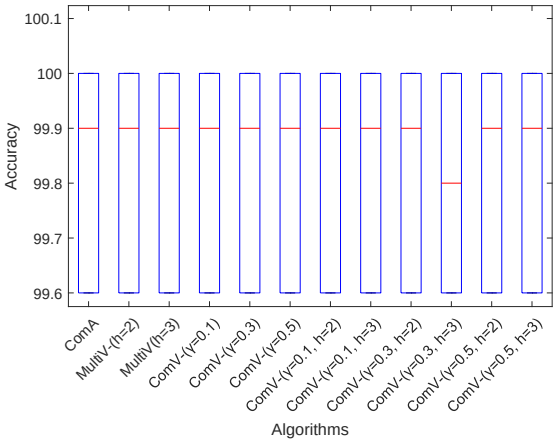


(d) p_{in}/p_{out}=1.5

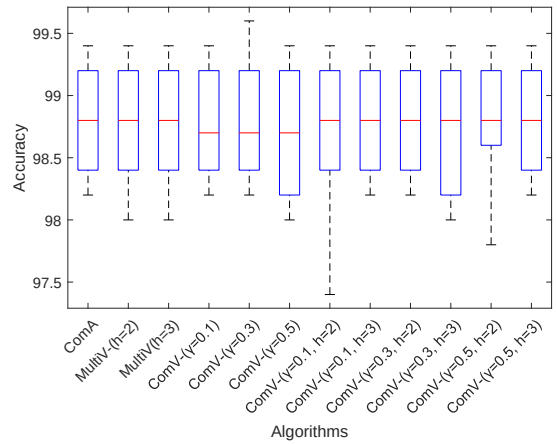
Figure 4.10: NMI of the experiments in the informative case on artificial graphs with k=3 layers



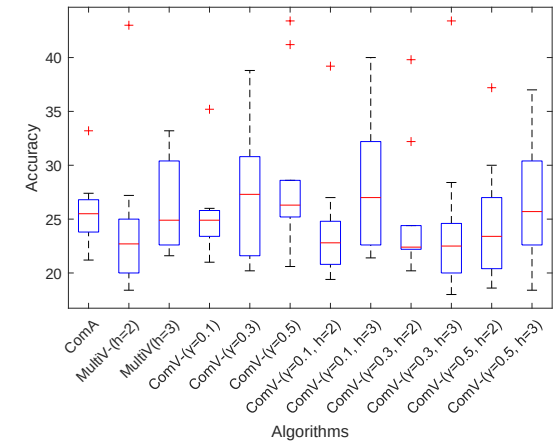
(a) p_{in}/p_{out}=3



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

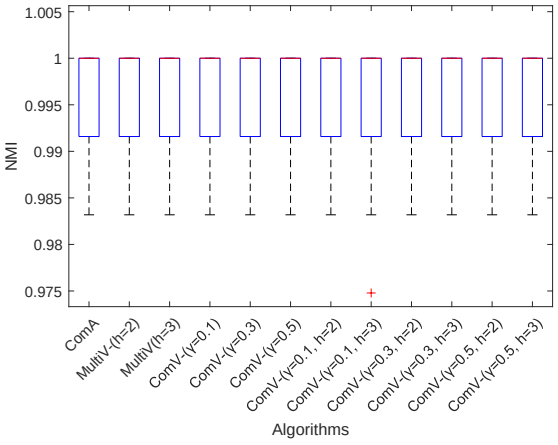


(c) p_{in}/p_{out}=2

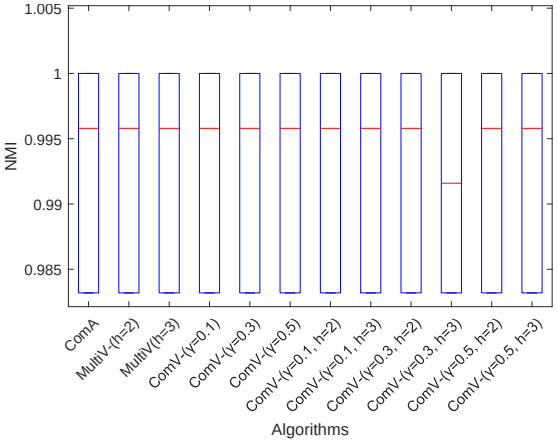


(d) p_{in}/p_{out}=1.5

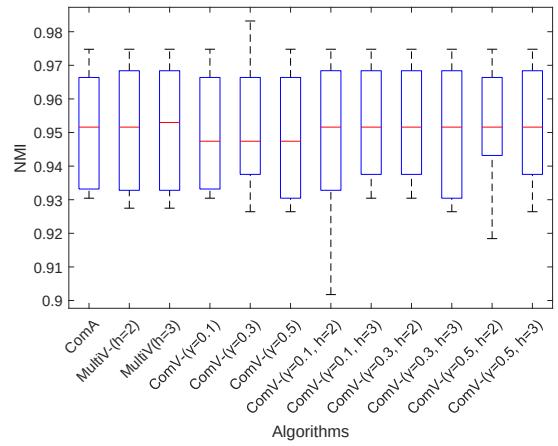
Figure 4.11: Accuracy of the experiments in the informative case on artificial graphs with k=3 layers



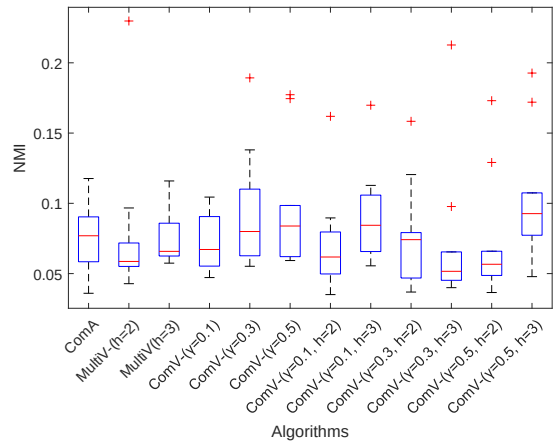
(a) p_{in}/p_{out}=3



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



(c) p_{in}/p_{out}=2

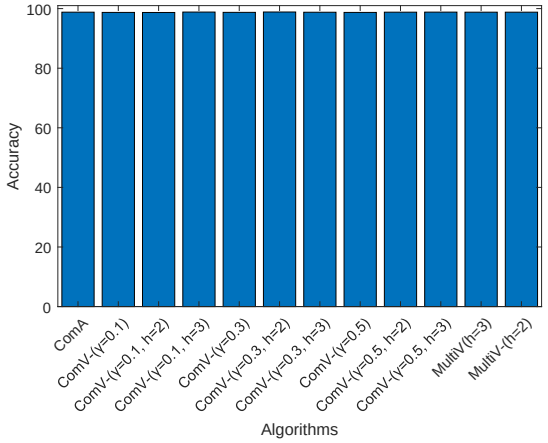


(d) p_{in}/p_{out}=1.5

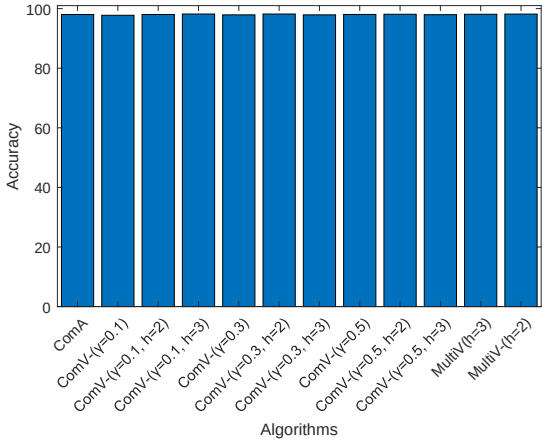
Figure 4.12: NMI of the experiments in the informative case on artificial graphs with k=3 layers

INFORMATIVE CASE: k=3												
	p_{in}/p_{out}=2			p_{in}/p_{out}=1.9			p_{in}/p_{out}=1.8			p_{in}/p_{out}=1.7		
	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu
ComA	98.80	0.952	5.42	98.02	0.924	6.04	96.20	0.865	6.59	61.88	0.440	7.56
MultiA h=2	98.80	0.952	233.60	98.16	**0.929**	249.74	90.72	0.804	275.88	65.76	0.467	293.36
MultiA h=3	98.80	0.952	357.67	98.10	0.927	396.72	84.92	0.742	417.78	**85.64**	**0.684**	440.85
ComV- γ=0.1	98.72	0.949	6.23	97.76	0.915	6.31	96.02	0.859	7.80	69.76	0.528	9.81
ComV- γ=0.3	98.74	0.950	6.36	97.90	0.920	7.07	89.36	0.791	8.47	52.44	0.342	9.20
ComV- γ=0.5	98.72	0.949	6.29	98.04	0.925	7.27	94.84	0.838	8.76	68.78	0.499	10.45
MultiV- γ=0.1, h=2	98.70	0.948	238.61	98.02	0.923	266.97	95.88	0.858	277.56	58.94	0.390	303.89
MultiV- γ=0.1, h=3	98.84	0.953	369.92	**98.18**	**0.929**	404.17	**96.30**	**0.868**	426.00	70.76	0.512	475.77
MultiV- γ=0.3, h=2	**98.86**	**0.954**	231.57	**98.18**	**0.929**	265.74	93.06	0.819	298.45	69.80	0.530	312.52
MultiV- γ=0.3, h=3	98.78	0.952	365.66	97.90	0.919	425.64	89.24	0.784	437.12	63.30	0.460	449.65
MultiV- γ=0.5, h=2	98.80	0.952	241.91	98.10	0.926	267.20	96.10	0.863	275.88	53.40	0.349	277.64
MultiV- γ=0.5, h=3	98.82	0.953	356.77	97.96	0.922	392.99	84.18	0.718	422.18	64.54	0.439	437.57

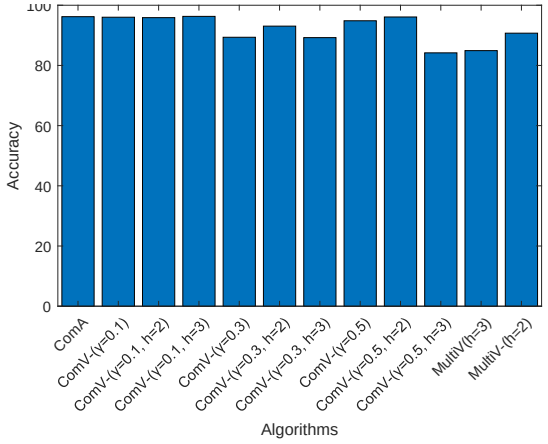
Table 4.4: Experiments in the informative case on artificial graphs with k=3 layers. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.



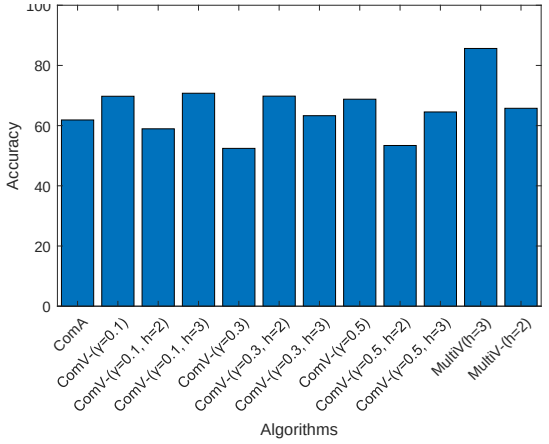
(a) p_{in}/p_{out}=2



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

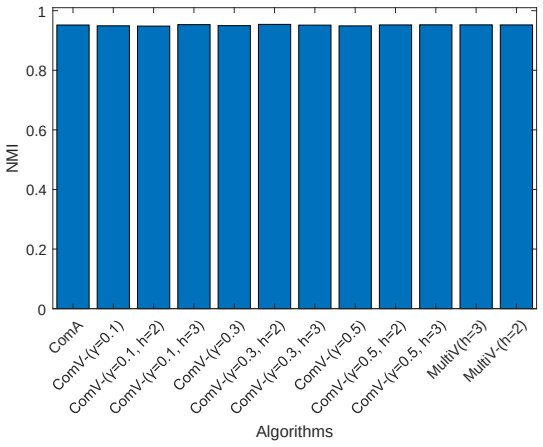


(c) p_{in}/p_{out}=1.8

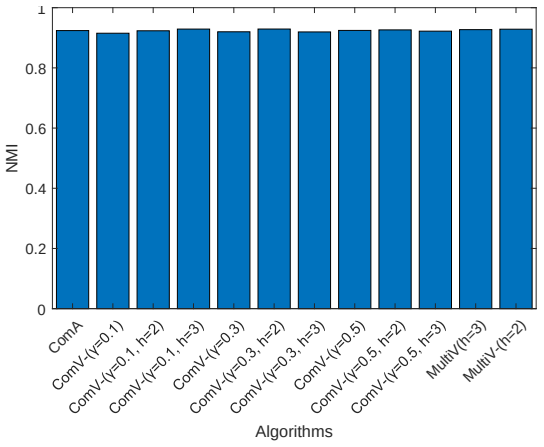


(d) p_{in}/p_{out}=1.7

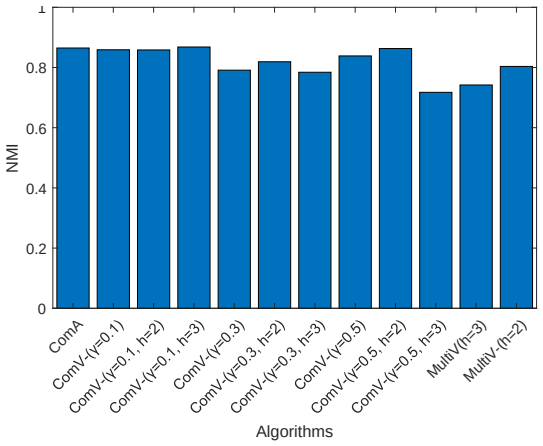
Figure 4.13: Accuracy of the experiments in the informative case on artificial graphs with k=3 layers



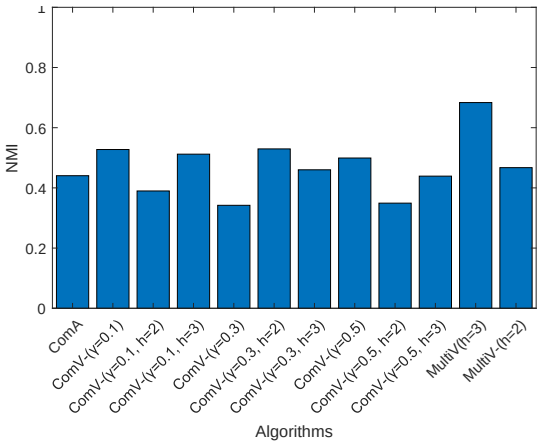
(a) p_{in}/p_{out}=2



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

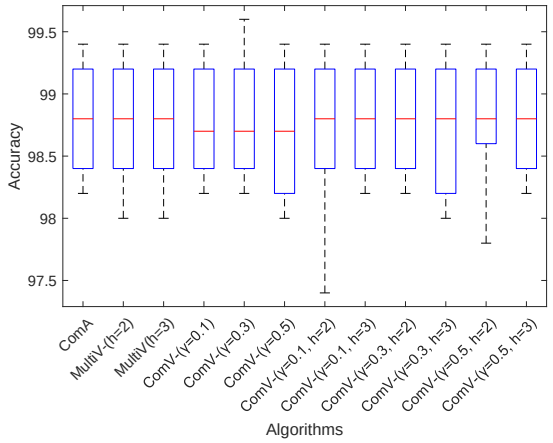


(c) p_{in}/p_{out}=1.8

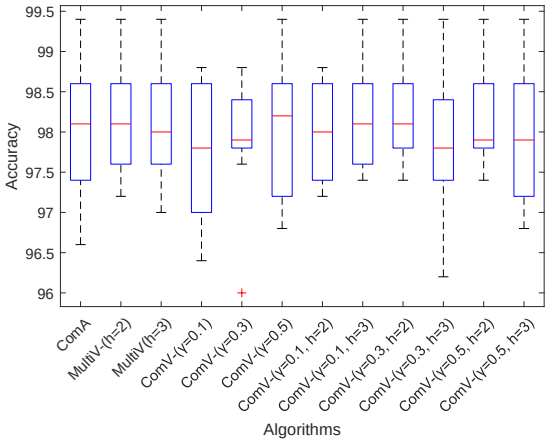


(d) p_{in}/p_{out}=1.7

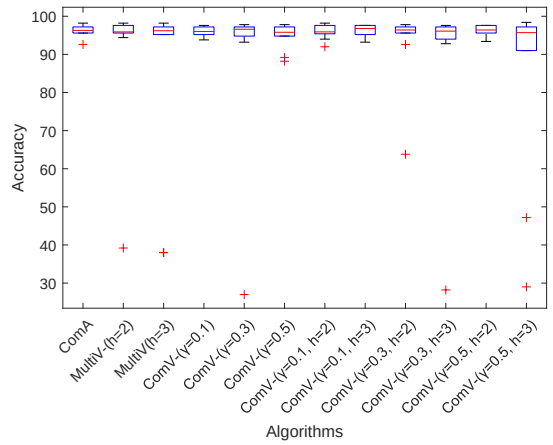
Figure 4.14: NMI of the experiments in the informative case on artificial graphs with k=3 layers



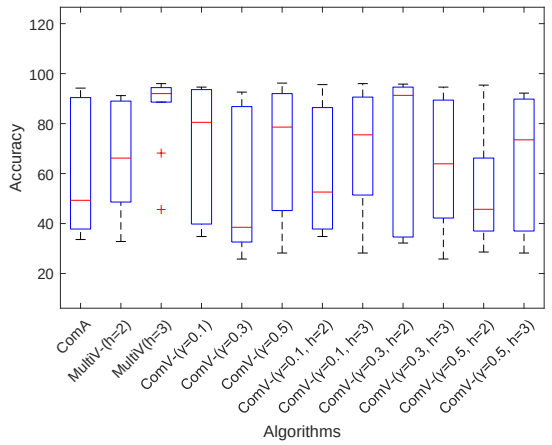
(a) p_{in}/p_{out}=2



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

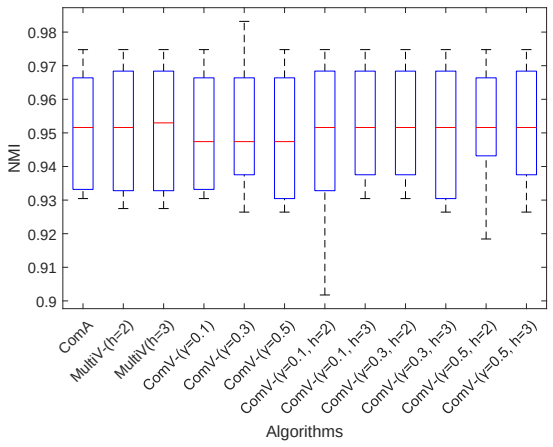


(c) p_{in}/p_{out}=1.8

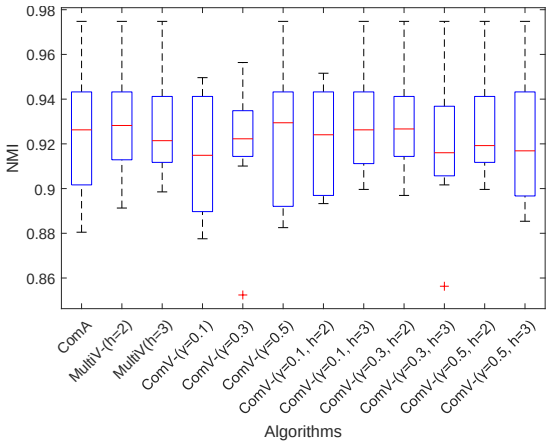


(d) p_{in}/p_{out}=1.7

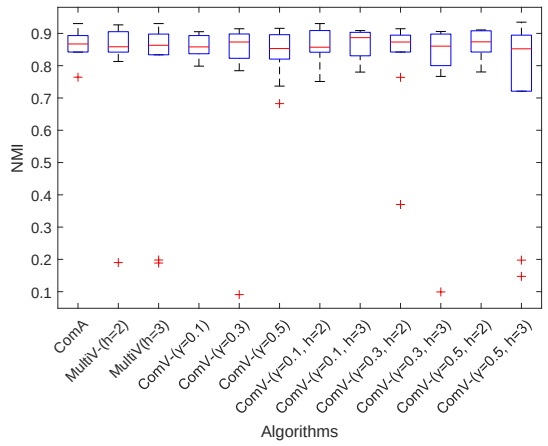
Figure 4.15: Accuracy of the experiments in the informative case on artificial graphs with k=3 layers



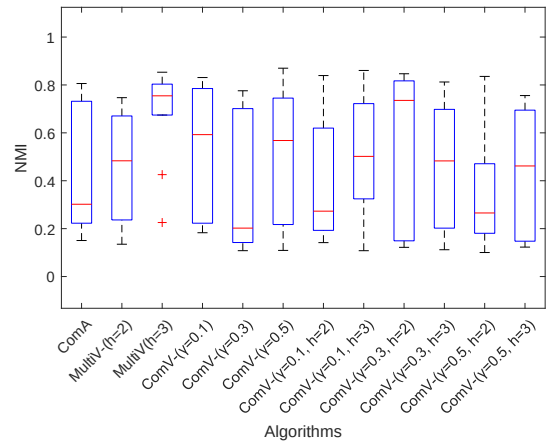
(a) p_{in}/p_{out}=2



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



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



(d) p_{in}/p_{out}=1.7

Figure 4.16: 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_{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.
The results get worse as the ratio between p_{in} and p_{out} decreases, in fact intuitively the communities are less defined when p_{out} is close to p_{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 $p_{in}/p_{out} = 3, 2.5$. All the algorithm gives bad results in the last case with $p_{in}/p_{out} = 2$.

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_{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.
MultiV+ with $\gamma = 0.5$, $h = 3$ achieves the best result in the first situation with $p_{in}/p_{out} = 3$. 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).

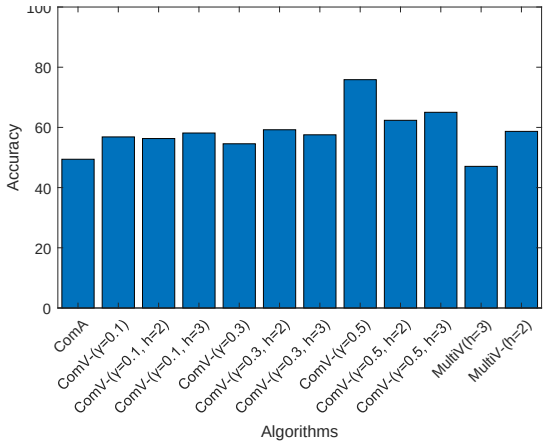
All the algorithms achieve better results in the informative case then the noisy one. Moreover, execution times are lower in the informative case than in the noisy case. In fact, in the informative case each layer has a piece of meaningful information, instead in the noisy case some layers can give wrong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

4.3 Real World Networks

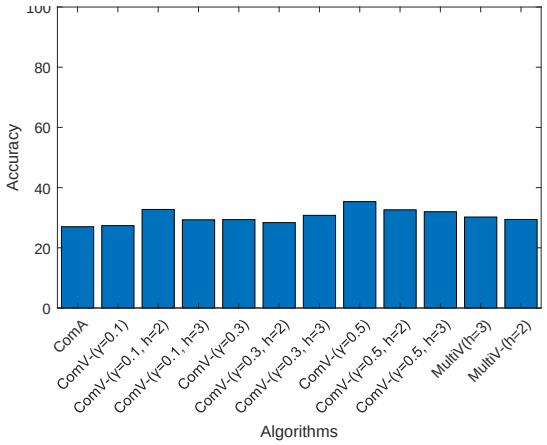
In this Section, we compare the performances of the proposed approaches on real networks. We considered three real datasets: 3sources [94], BBCSports [95] and Wikipedia [96]. We used the corresponding layer matrices provided by Mercado et al. [97], which have used a similarity measure and have considered 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

NOISY CASE: k=2									
	p_{in}/p_{out}=3			p_{in}/p_{out}=2.5			p_{in}/p_{out}=2		
	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu
ComA	49.42	0.318	7.34	27.00	0.109	7.35	22.46	0.054	8.01
MultiA h=2	58.68	0.369	255.28	29.42	0.116	289.26	21.06	0.053	312.98
MultiA h=3	47.06	0.289	226.51	30.22	0.128	383.26	21.62	0.056	483.20
ComV+ γ=0.1	56.84	0.382	7.03	27.36	0.102	8.32	21.30	0.053	9.00
ComV+ γ=0.3	54.56	0.343	5.56	29.36	0.128	9.52	21.50	0.054	9.99
ComV+ γ=0.5	**75.86**	**0.581**	9.19	**35.34**	**0.173**	9.62	21.88	0.058	10.41
MultiV+ γ=0.1, h=2	56.30	0.368	260.59	32.74	0.142	281.59	20.78	0.054	300.20
MultiV+ γ=0.1, h=3	58.14	0.370	379.66	29.30	0.126	386.27	22.08	0.060	439.44
MultiV+ γ=0.3, h=2	59.22	0.396	190.88	28.36	0.126	254.34	21.30	0.052	269.15
MultiV+ γ=0.3, h=3	57.54	0.394	211.16	30.78	0.144	379.35	21.36	0.057	525.10
MultiV+ γ=0.5, h=2	62.36	0.432	164.39	32.62	0.157	252.06	**24.02**	**0.067**	283.00
MultiV+ γ=0.5, h=3	65.02	0.466	318.62	31.98	0.154	407.74	22.12	**0.067**	474.81

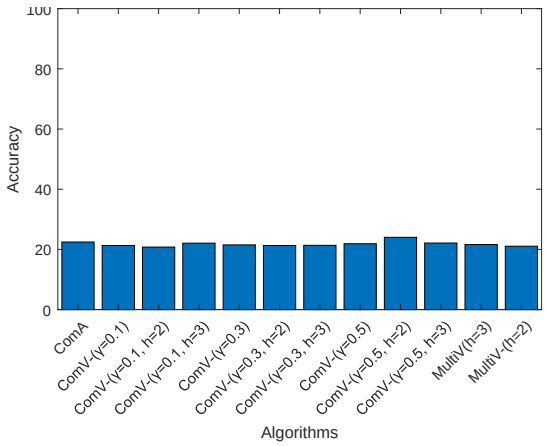
Table 4.5: Experiments in the noisy case on artificial graphs with k=2 layers. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.



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

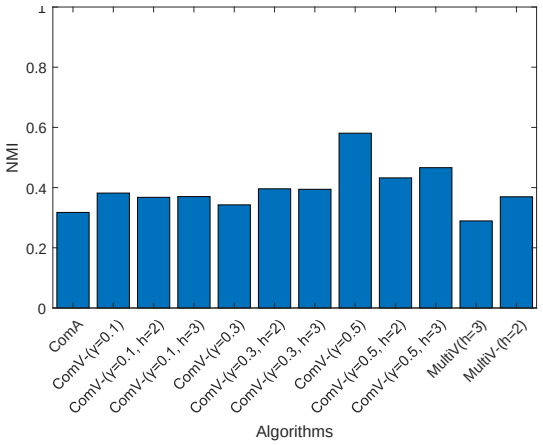


(b) p_{in}/p_{out}=2.5

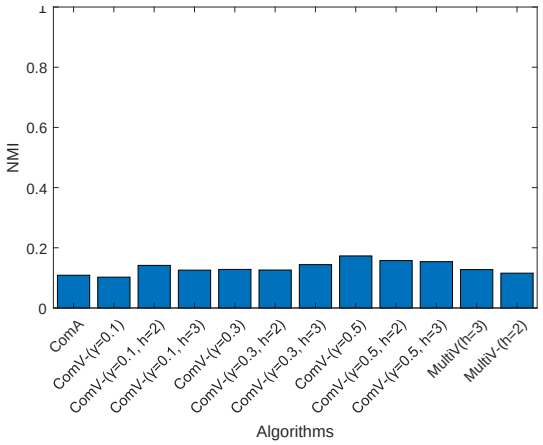


(c) p_{in}/p_{out}=2

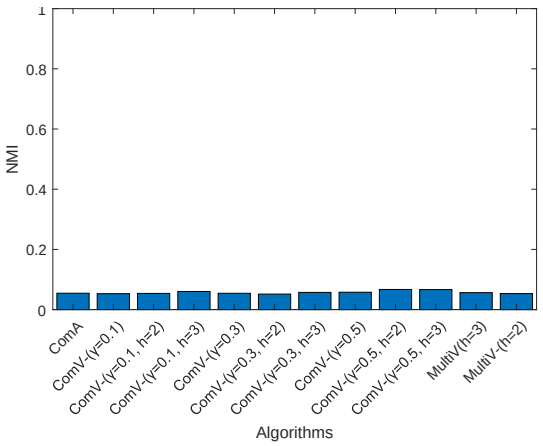
Figure 4.17: Accuracy of the experiments in the noisy case on artificial graphs with k=2 layers



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

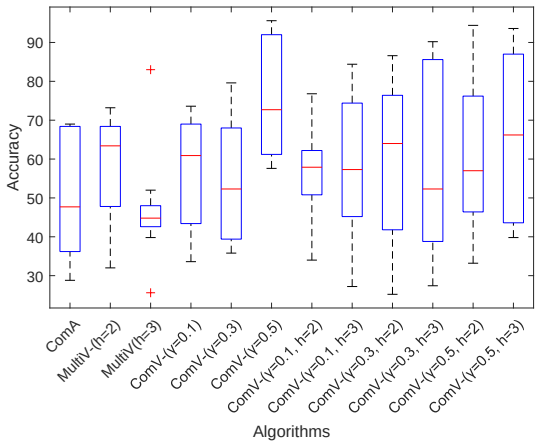


(b) p_{in}/p_{out}=2.5

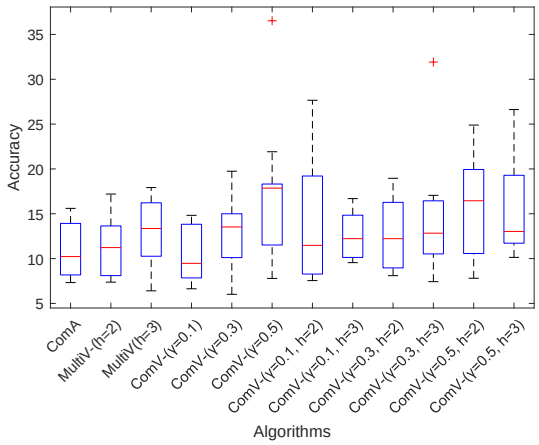


(c) p_{in}/p_{out}=2

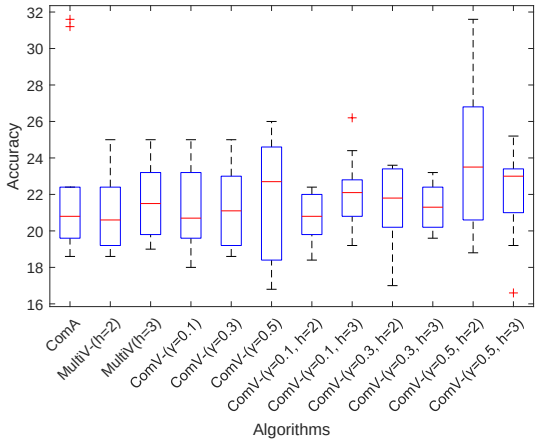
Figure 4.18: NMI of the experiments in the noisy case on artificial graphs with k=2 layers



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

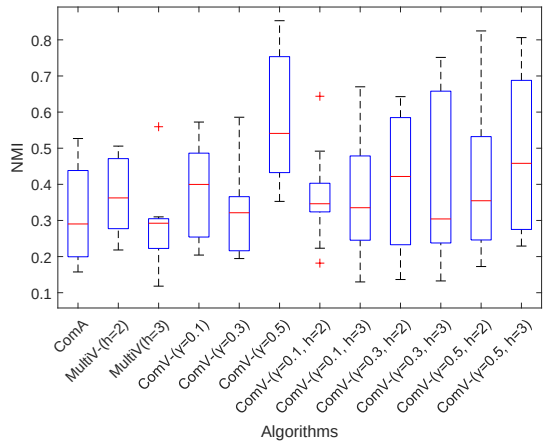


(b) p_{in}/p_{out}=2.5

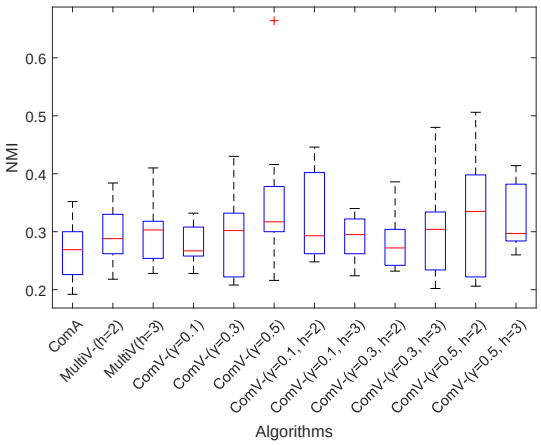


(c) p_{in}/p_{out}=2

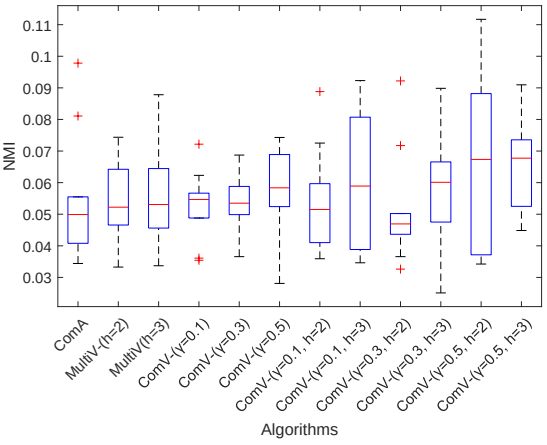
Figure 4.19: Accuracy of the experiments in the noisy case on artificial graphs with k=2 layers



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



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

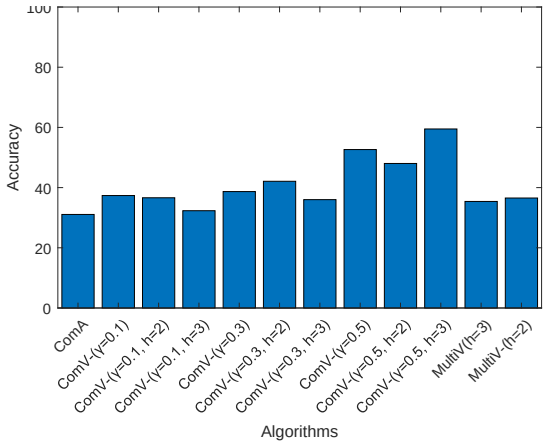


(c) p_{in}/p_{out}=2

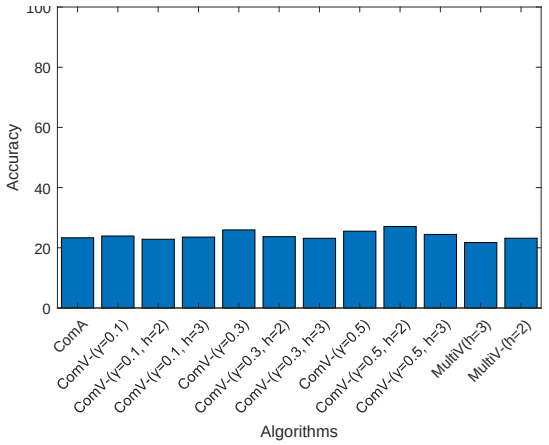
Figure 4.20: NMI of the experiments in the noisy case on artificial graphs with k=2 layers

NOISY CASE: k=3									
	p_{in}/p_{out}=3			p_{in}/p_{out}=2.5			p_{in}/p_{out}=2		
	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu
ComA	31.08	0.140	6.61	23.32	0.074	7.15	21.48	0.046	8.23
MultiA h=2	36.54	0.183	198.97	23.18	0.069	242.43	20.72	0.045	280.91
MultiA h=3	35.40	0.169	366.69	21.76	0.066	431.68	**21.86**	0.054	441.05
ComV+ γ=0.1	37.36	0.196	8.67	23.92	0.085	9.97	20.34	0.044	13.69
ComV+ γ=0.3	38.68	0.219	8.58	25.94	0.095	9.57	20.76	0.045	11.57
ComV+ γ=0.5	52.64	0.328	9.88	25.52	0.092	10.34	21.32	0.055	12.94
MultiV+ γ=0.1, h=2	36.62	0.188	245.82	22.84	0.070	293.56	21.06	0.037	331.74
MultiV+ γ=0.1, h=3	32.32	0.151	348.44	23.56	0.077	363.38	21.52	0.051	440.29
MultiV+ γ=0.3, h=2	42.10	0.227	225.27	23.70	0.082	260.47	21.30	0.047	277.16
MultiV+ γ=0.3, h=3	36.00	0.189	373.14	23.16	0.069	418.36	21.20	0.042	439.75
MultiV+ γ=0.5, h=2	48.02	0.306	253.78	**27.08**	**0.102**	292.30	21.48	**0.057**	309.10
MultiV+ γ=0.5, h=3	**59.48**	**0.402**	362.97	24.44	0.099	408.00	20.80	0.049	442.13

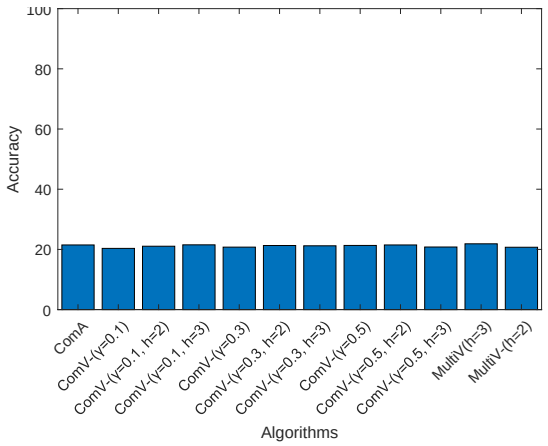
Table 4.6: Experiments in the noisy case on artificial graphs with k=3 layers. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.



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

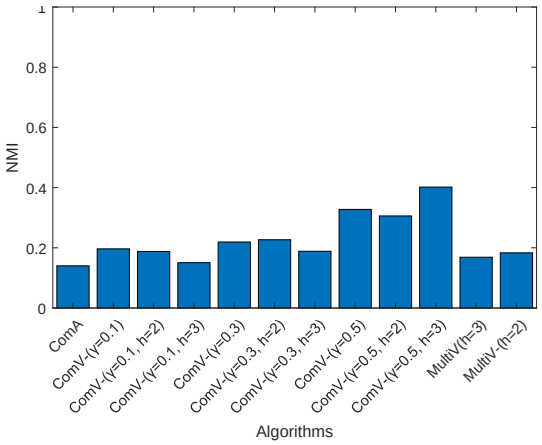


(b) p_{in}/p_{out}=2.5

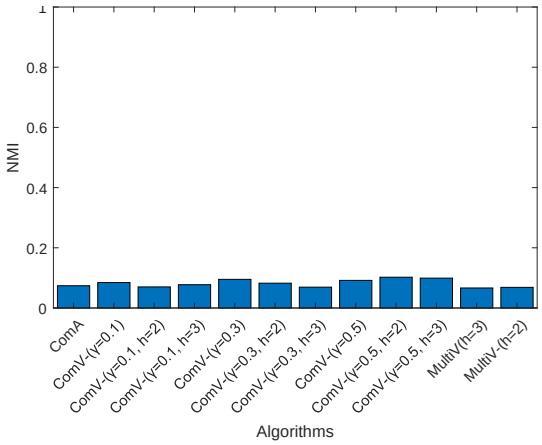


(c) p_{in}/p_{out}=2

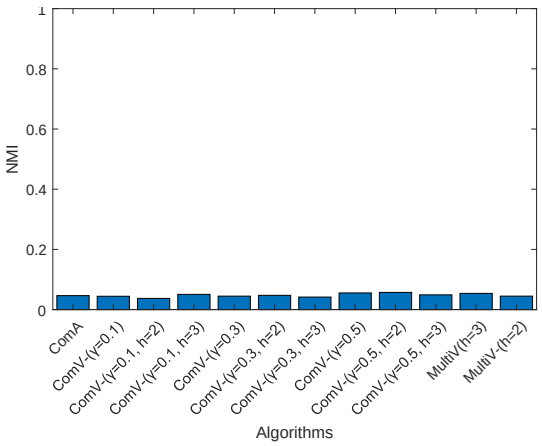
Figure 4.21: Accuracy of the experiments in the noisy case on artificial graphs with k=3 layers



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

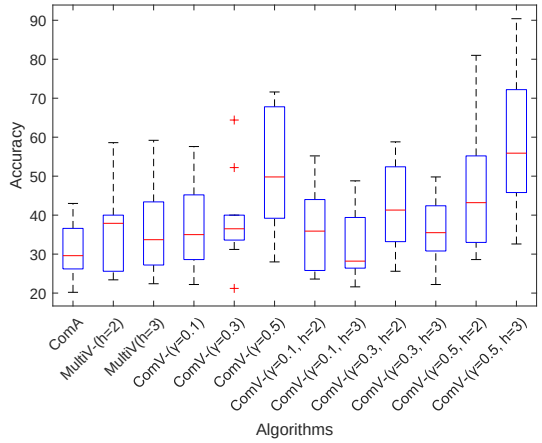


(b) p_{in}/p_{out}=2.5

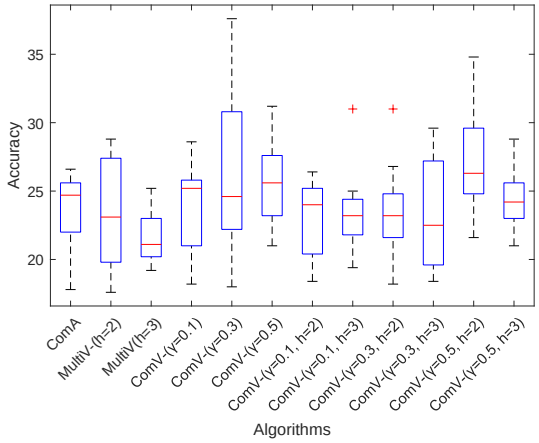


(c) p_{in}/p_{out}=2

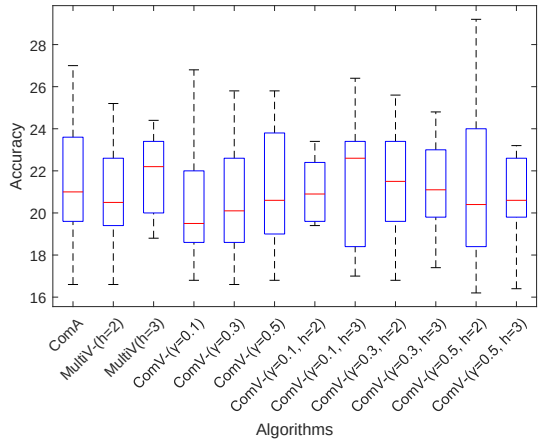
Figure 4.22: NMI of the experiments in the noisy case on artificial graphs with k=3 layers



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

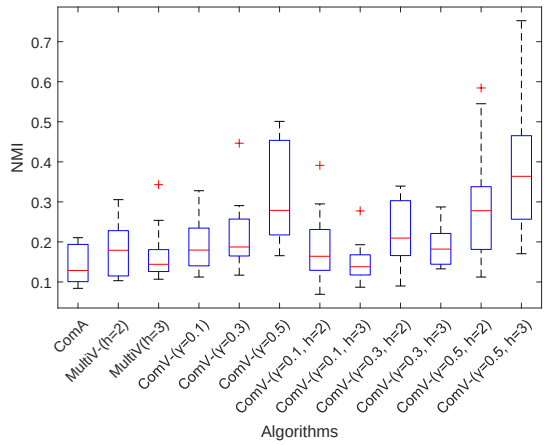


(b) p_{in}/p_{out}=2.5

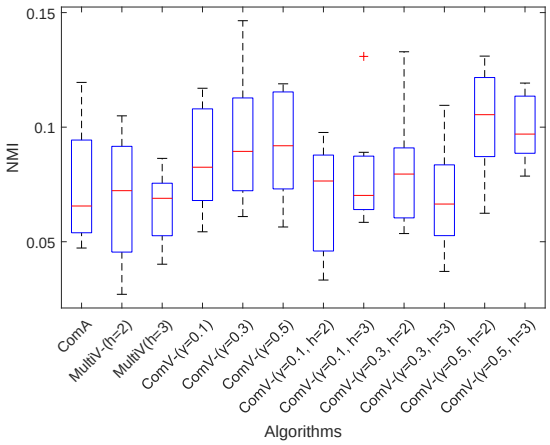


(c) p_{in}/p_{out}=2

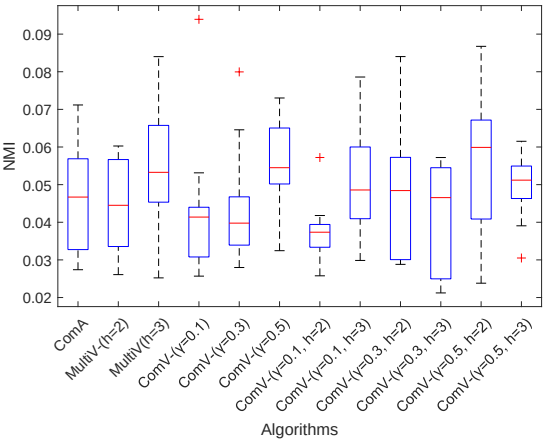
Figure 4.23: Accuracy of the experiments in the noisy case on artificial graphs with k=3 layers



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



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



(c) p_{in}/p_{out}=2

Figure 4.24: NMI of the experiments in the noisy case on artificial graphs with k=3 layers

muti-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, 85, 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.

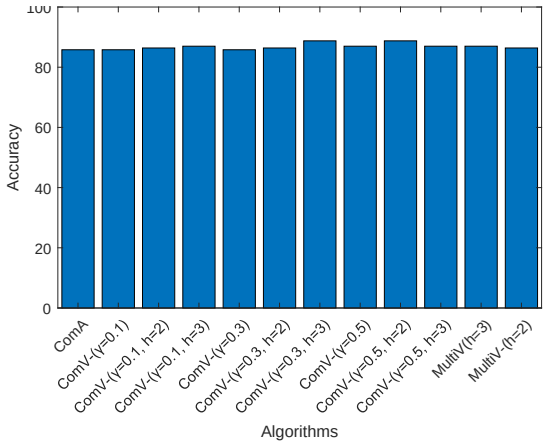
For this case, we compared the models *community-average* (ComA) and *community-variance-plus* (ComV+) (Section 3.1), *multi-average* (MultiA) and *multi-variance-plus* (MultiV+) (Section 3.2). The idea behind is that for the noisy case we would like to maximize both the average and the variance of modularity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

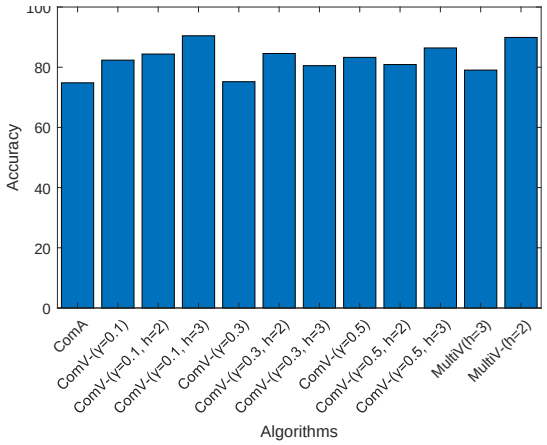
All the algorithms achieve better results in the informative case than the noisy one. Moreover, execution times are lower in the noisy case than in the informative case. In fact, in the informative case each layer has a piece of meaningful information, in fact in the noisy case some layers can give wrong information.
As seen in the informative case, execution times increase with increasing number of nodes and are higher for the Louvain Multiobjective methods.

INFORMATIVE CASE									
	3sources			BBCSport			Wikipedia		
	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu
ComA	85.80	0.749	0.21	74.82	0.753	2.09	55.56	0.544	4.02
MultiA h=2	86.39	0.765	3.59	89.89	0.825	51.44	52.53	0.521	136.54
MultiA h=3	86.98	0.773	5.30	79.04	0.791	85.77	**55.84**	0.546	207.34
ComV- γ=0.1	85.80	0.749	0.25	82.35	0.789	2.54	54.83	0.520	5.87
ComV- γ=0.3	85.80	0.749	0.26	75.18	0.751	2.40	41.70	**0.285**	5.43
ComV- γ=0.5	86.98	0.781	0.24	83.27	0.798	2.69	37.52	0.285	8.94
MultiV- γ=0.1, h=2	86.39	0.765	3.69	84.38	0.784	56.48	54.55	0.503	144.25
MultiV- γ=0.1, h=3	86.98	0.773	5.60	**90.44**	**0.837**	89.38	54.40	0.520	265.18
MultiV- γ=0.3, h=2	86.39	0.765	3.64	84.56	0.787	56.22	44.30	0.337	167.55
MultiV- γ=0.3, h=3	**88.76**	0.805	5.97	80.51	0.787	70.36	44.30	0.369	264.88
MultiV- γ=0.5, h=2	**88.76**	**0.812**	4.13	80.88	0.784	50.78	37.09	0.266	178.91
MultiV- γ=0.5, h=3	86.98	0.775	8.12	86.40	0.816	82.10	36.65	0.279	207.46

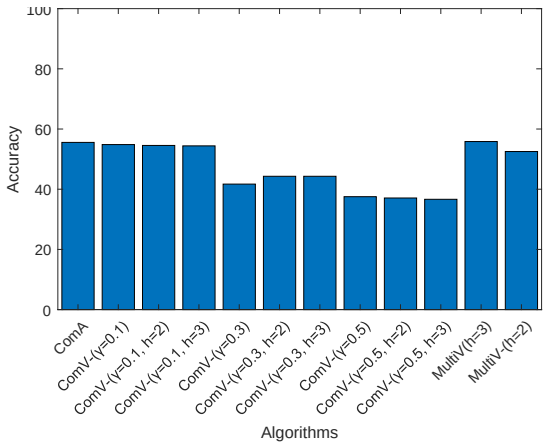
Table 4.7: Experiments in the informative case on real datasets. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.



(a) 3sources

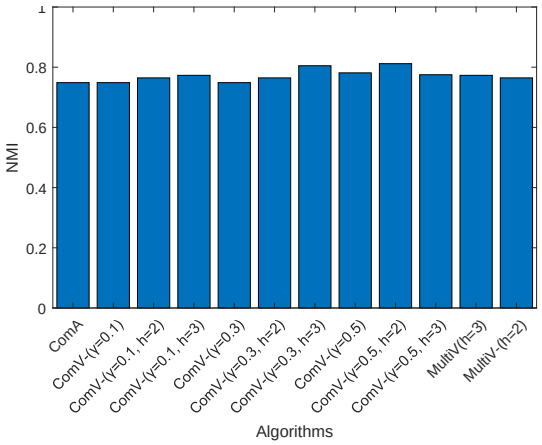


(b) BBCSport

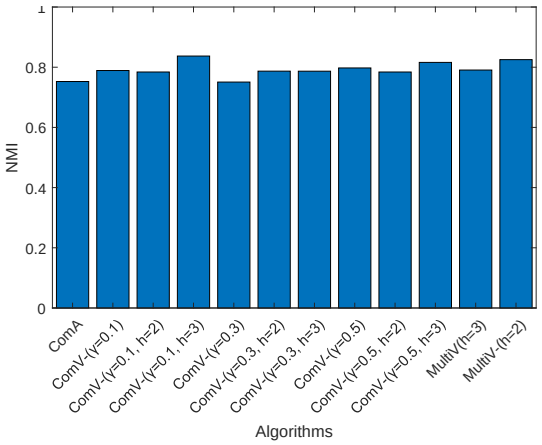


(c) Wikipedia

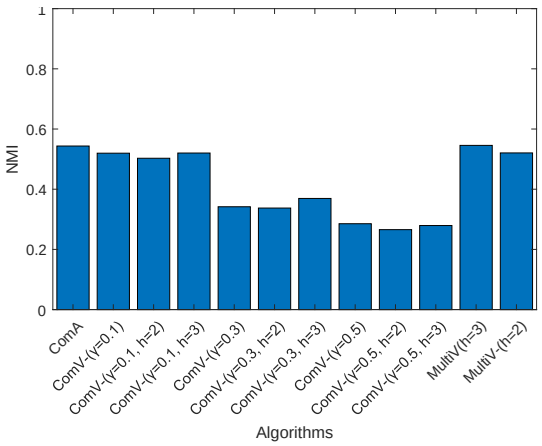
Figure 4.25: Accuracy of the experiments in the informative case on real datasets



(a) 3sources



(b) BBCSport

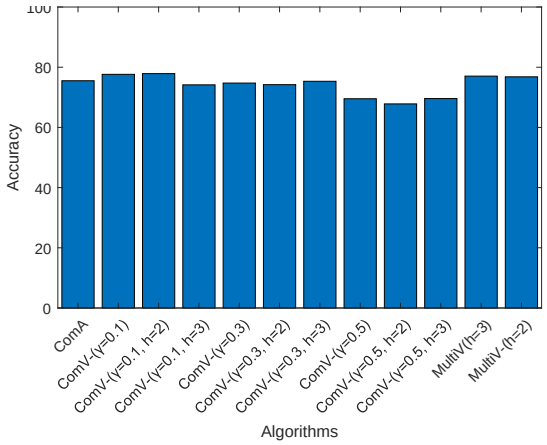


(c) Wikipedia

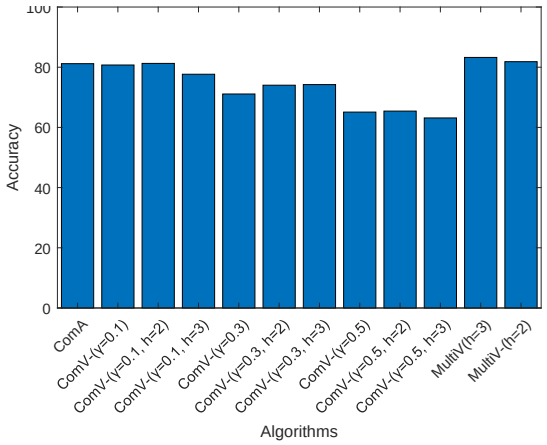
Figure 4.26: NMI of the experiments in the informative case on real datasets

NOISY CASE									
	3sources			BBCSport			Wikipedia		
	Ac	NMI	Cpu	Ac	NMI	Cpu	Ac	NMI	Cpu
ComA	75.50	0.706	1.01	81.18	0.749	4.49	17.59	0.065	7.84
MultiA h=2	76.80	0.715	14.48	81.84	0.752	128.82	17.30	0.067	397.11
MultiA h=3	77.04	**0.717**	22.32	**83.25**	**0.757**	192.67	17.34	**0.068**	452.51
ComV+ γ=0.1	77.63	**0.717**	1.20	80.74	0.739	5.29	17.29	0.063	8.79
ComV+ γ=0.3	74.73	0.698	1.28	71.10	0.686	4.97	17.17	0.061	9.22
ComV+ γ=0.5	69.53	0.661	1.09	65.09	0.648	5.03	16.26	0.058	9.07
MultiV+ γ=0.1, h=2	**77.87**	**0.717**	15.00	81.27	0.739	129.70	**17.84**	0.066	302.08
MultiV+ γ=0.1, h=3	74.14	0.698	23.33	77.67	0.728	190.31	17.71	0.066	453.98
MultiV+ γ=0.3, h=2	74.20	0.699	14.69	74.03	0.698	122.65	17.55	0.063	317.09
MultiV+ γ=0.3, h=3	75.33	0.703	21.85	74.23	0.699	192.38	17.30	0.062	441.93
MultiV+ γ=0.5, h=2	67.81	0.658	14.25	65.42	0.648	125.40	16.70	0.058	320.44
MultiV+ γ=0.5, h=3	69.59	0.661	21.05	63.14	0.644	187.00	17.11	0.058	450.51

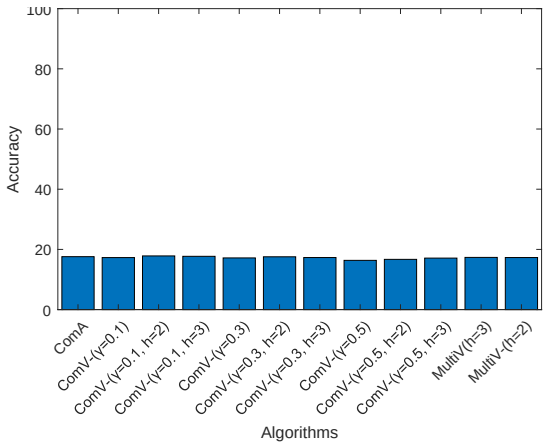
Table 4.8: Experiments in the noisy case on real datasets. Notation: **best** performances are marked with bold fonts and gray background and second best performances with only gray background.



(a) 3sources

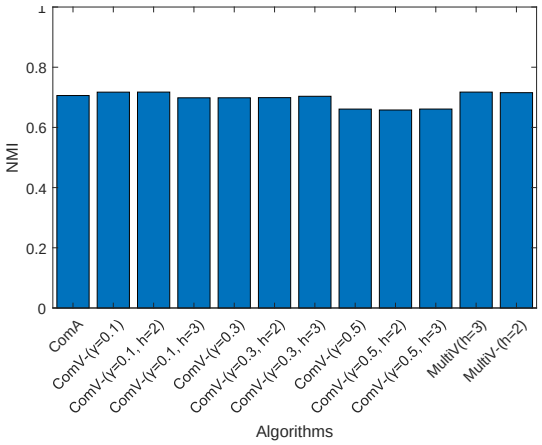


(b) BBCSport

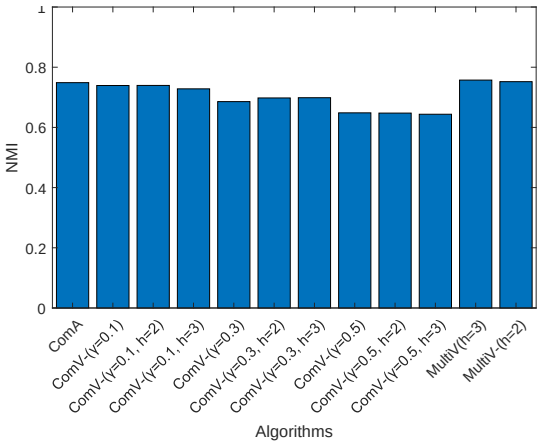


(c) Wikipedia

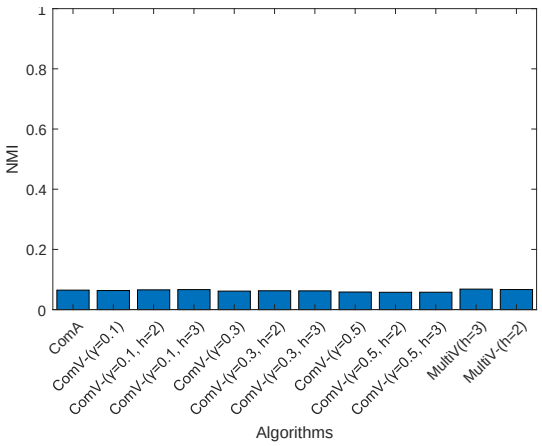
Figure 4.27: Accuracy of the experiments in the noisy case on real datasets



(a) 3sources

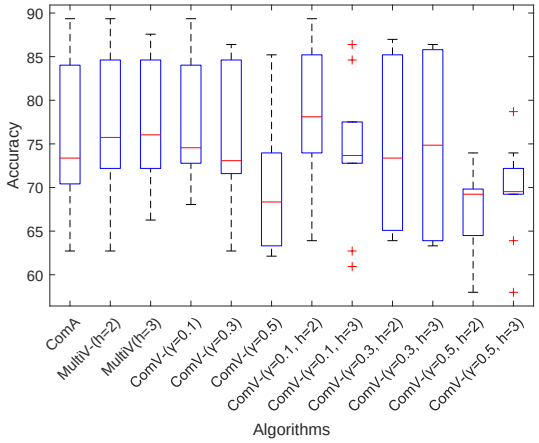


(b) BBCSport

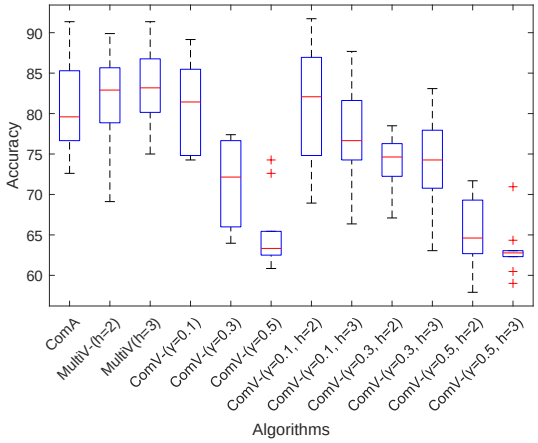


(c) Wikipedia

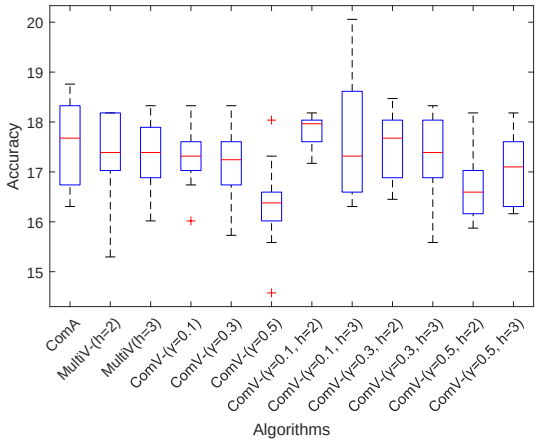
Figure 4.28: NMI of the experiments in the noisy case on real datasets



(a) 3sources

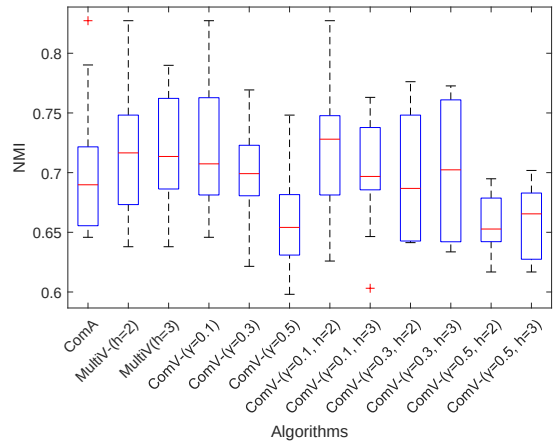


(b) BBCSport

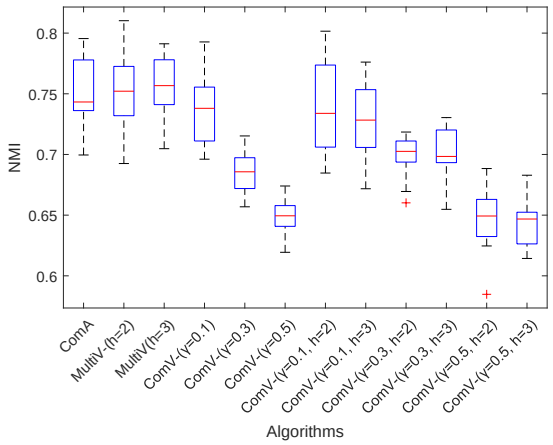


(c) Wikipedia

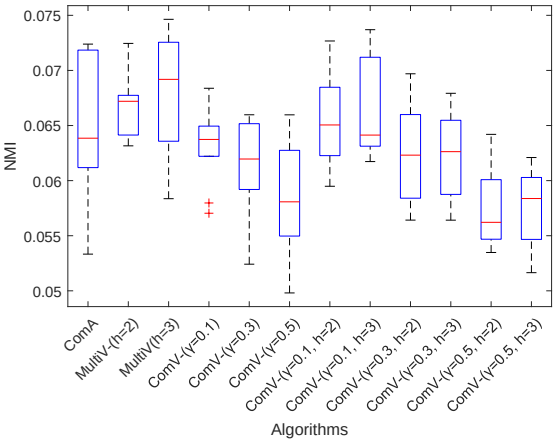
Figure 4.29: Accuracy of the experiments in the noisy case on real datasets



(a) 3sources



(b) BBCSport



(c) Wikipedia

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 γ 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
%   ending :
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
```

```
29      k = S(3);
30  else k = 1;
31  end
32
33  ending = 0;
34
35  for s=1:k
36      M2(:,:,s)=M(:,:,s)-diag(diag(M(:,:,s)));
37  end
38
39  %total weight of the graph
40  for s=1:k
41      if z==1
42          M(:,:,s) = M(:,:,s) + diag(diag(M(:,:,s)));
43      end
44      m{s} = sum(sum(M(:,:,s)))/2;
45  end
46
47  Niter = 0; %number of iterations
48
49  if (sum(cellfun(@(x)x>0,m))==0) | N == 1
50    ending = 1;
51    COMTY = 0;
52    return
53  end
54
55  %Delate layers with m=0
56  M(:,:,(cellfun(@(x)x==0,m)))=[];
57  k = k - sum(cellfun(@(x)x==0,m));
58
59  COM = 1:S(1); % Community of node i
60
61  for s=1:k
62      K(:,:,s) = sum(M(:,:,s)); % K(i)= sum of wieght incident to node i
63      SumTot(:,:,s) = sum(M(:,:,s)); %SumTot(i)= sum of weight incident to
            nodes in community i
64      SumIn(:,:,s) = diag(M(:,:,s))'; %SumIn(i)= sum of weight inside
           community i (loops)
65                                      %At the beginning each node is a
                                           community
66      Q{s} = compute_modularity(COM,M(:,:,s));
67  end
68  Average = sum([Q{:}])/k;
69
70  %Neighbor{j}{s} neighbor of node j in layer s
71  for j=1:N
72      for s=1:k
73          temp=M2(:,:,s);
74          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;
    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(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
        %gain average
        for s=1:k
            DQ_j{s} = GQ_i{s} + GQ_j{s};
        end
        G(Cj) =(sum([DQ_j{:}])/k);
        if G(Cj) > best_increase
          best_increase = G(Cj);
          DQ_j_t = DQ_j;
          Cnew_t = Cj;
        end
      end
    end
```

```
119     if best_increase > -10^(-15)
120       Cnew = Cnew_t;
121       Average = Average + best_increase;
122       for s=1:k
123           Q{s} = Q{s} + DQ_j_t{s};
124       end
125     end
126     %Recalculate
127     Ck = (COM==Cnew);
128     for s=1:k
129         SumIn(:,Cnew,s) = SumIn(:,Cnew,s) + 2*sum(M(i,Ck,s)) + M(i,i,s);
130         SumTot(:,Cnew,s) = SumTot(:,Cnew,s) + K(:,i,s);
131     end
132     COM(i) = Cnew;
133     if (Cnew ~= Ci)
134       gain = 1;
135     end
136   end
137   Niter = Niter + 1;
138 end
139 Niter = Niter - 1;
140 [COM] = reindex_com(COM);
141 COMTY.COM{1} = COM;
142 COMTY.MOD{1} = [Q{:}];
143 COMTY.Average(1) = Average;
144 COMTY.Niter(1) = Niter;
145
146
147 % Perform part 2
148 if (z == 1)
149   Mnew = M;
150   Mold = Mnew;
151   COMcur = COM;
152   COMfull = COM;
153   j = 2;
154   while 1
155     Mold = Mnew;
156     S2 = size(Mold);
157     Nnode = S2(1);
158
159     COMu = unique(COMcur);
160     Ncom = length(COMu);
161     ind_com = sparse(Ncom,Nnode);
162     ind_com_full = sparse(Ncom,N);
163     for p=1:Ncom
164       ind = find(COMcur==p);
165       ind_com(p,1:length(ind)) = ind;
166     end
167     for p=1:Ncom
```

```
168         ind = find(COMfull==p);
169         ind_com_full(p,1:length(ind)) = ind;
170       end
171       Mnew = [];
172       for s=1:k
173           Mnew(:,:,s) = zeros(Ncom,Ncom); %new matrix (each node  is a
                community)
174           for m=1:Ncom
175               for n=m:Ncom
176                   ind1 = ind_com(m,:);
177                   ind2 = ind_com(n,:);
178                   %weights of edges between communities
179                    Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)
                        ));
180                    Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)
                        ));
181               end
182           end
183       end
184       [COMt e] = community_average(Mnew,0);
185       if (e ~= 1)
186         COMfull = sparse(1,N);
187         COMcur = COMt.COM{1};
188         for p=1:Ncom
189           ind1 = ind_com_full(p,:);
190           COMfull(ind1(ind1>0)) = COMcur(p);
191         end
192         [COMfull2] = reindex_com(COMfull);
193         COMTY.COM{j} = COMfull2;
194         COMTY.MOD{j} = COMt.MOD{1};
195         COMTY.Average(j) = COMt.Average(1);
196         COMTY.Niter(j) = COMt.Niter;
197         Ind = (COMfull2 == COMTY.COM{j-1});
198         if (sum(Ind) == length(Ind))
199           return;
200         end
201       else
202         return;
203       end
204       j = j + 1;
205     end
206   end
207   end
208
209   % Re-index community partition by size
210   function [C Ss] = reindex_com(COMold)
211   C = sparse(1,length(COMold));
212   COMu = unique(COMold);
213   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 modulartiy
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)

```
%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 :
% 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_minus(M,lambda)

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

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

if nargin < 3
```

```
29      z = 1;
30  end
31
32  S = size(M);
33  N = S(1);
34  if length(S)==3
35      k = S(3);
36  else k = 1;
37  end
38
39  ending = 0;
40
41  for s=1:k
42      M2(:,:,s)=M(:,:,s)-diag(diag(M(:,:,s)));
43  end
44
45  %total weight of the graph
46  for s=1:k
47      if z==1
48          M(:,:,s) = M(:,:,s) + diag(diag(M(:,:,s)));
49      end
50      m{s} = sum(sum(M(:,:,s)))/2;
51  end
52
53  Niter = 0; %number of iterations
54
55  if (sum(cellfun(@(x)x>0,m))==0) | N == 1
56    ending = 1;
57    COMTY = 0;
58    return;
59  end
60
61  %Delate layers with m=0
62  M(:,:,(cellfun(@(x)x==0,m)))=[];
63  k = k - sum(cellfun(@(x)x==0,m));
64
65  COM = 1:S(1); % Community of node i
66
67  for s=1:k
68      K(:,:,s) = sum(M(:,:,s)); % K(i)= sum of wieght incident to node i
69      SumTot(:,:,s) = sum(M(:,:,s)); %SumTot(i)= sum of weight incident to
            nodes in community i
70      SumIn(:,:,s) = diag(M(:,:,s))'; %SumIn(i)= sum of weight inside
           community i (loops)
71                                        %At the beginning each node is a
                                             community
72      Q{s} = compute_modularity(COM,M(:,:,s));
73  end
74  Average = sum([Q{:}])/k;
```

```
75 if k==1
76     Variance = 0;
77 else
78     Variance = (sum(([Q{:}]-Average).^(2))) / (k-1); %variance
79 end
80 Function = (1-lambda) * Average - lambda * Variance; %function
81
82 %Neighbor{j}{s} neighbor of node j in layer s
83 for j=1:N
84     for s=1:k
85         temp=M2(:,:,s);
86         Neighbor_j{s} = find(temp(j,:));
87     end
88     Neighbor{j}=Neighbor_j;
89 end
90
91 gain = 1;
92 while (gain == 1)
93   gain = 0;
94   for i=1:N
95     Ci = COM(i);
96     NB=unique(cat(2,Neighbor{i}{:}));
97     G = zeros(1,N); % Gain vector
98     best_increase = -inf;
99     Cnew = Ci;
100    COM(i) = -1;
101    %remove i from its community
102    for s=1:k
103        CNi = (COM==Ci); %list of nodes in Ci community, without i
104        Ki_in_i{s} = sum(M(i,CNi,s)); %sum of weights between i and Ci
105        %Gain of modularity
106        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)));
107        %Recalculate values
108        SumTot(:,Ci,s) = SumTot(:,Ci,s) - K(:,i,s); %weights incident to
               Ci community
109        SumIn(:,Ci,s) = SumIn(:,Ci,s) - 2*sum(M(i,CNi,s)) - M(i,i,s); %
               weights community i
110    end
111    for j=1:length(NB)
112      Cj = COM(NB(j));
113      if (G(Cj) == 0)
114        CNj = (COM==Cj); %nodes in community Cj, without j
115        for s=1:k
116            Ki_in_j{s} = sum(M(i,CNj,s)); %sum of weights between i and
                   Cj
117            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
118        end
```

```
119         %variance gain
120         if k==1
121             GV_j = 0;
122         else
123             for s=1:k
124                 DQ{s} = GQ_i{s} + GQ_j{s}; %gains
125             end
126             M_DQ = sum([DQ{:}])/k; %gain average
127             GV_j = ((sum(([DQ{:}] - M_DQ).^(2))) / (k-1)) + (2/(k-1)) *
                    sum(sum(([Q{:}]-Average) .* ([DQ{:}] - M_DQ))); %gain
                    variance
128         end
129         %gain of the function
130         G(Cj) = (1-lambda)*M_DQ - lambda* GV_j;
131         if G(Cj) > best_increase
132           best_increase = G(Cj); %gain function
133           Q_t = DQ; %gain of modularity;
134           M_t = M_DQ; %gain average
135           V_t = GV_j;%gain of variance
136           Cnew_t = Cj;
137         end
138       end
139     end
140     if best_increase > -10^(-15)
141       Cnew = Cnew_t;
142       for s=1:k
143           Q{s} = Q{s} + Q_t{s};
144       end
145       Average = Average + M_t;
146       Variance = Variance + V_t;
147       Function = Function + best_increase;
148     end
149     %Recalculate
150     Ck = (COM==Cnew);
151     for s=1:k
152         SumIn(:,Cnew,s) = SumIn(:,Cnew,s) + 2*sum(M(i,Ck,s)) + M(i,i,s);
153         SumTot(:,Cnew,s) = SumTot(:,Cnew,s) + K(:,i,s);
154     end
155     COM(i) = Cnew;
156     if (Cnew ~= Ci)
157       gain = 1;
158     end
159   end
160   Niter = Niter + 1;
161 end
162 Niter = Niter - 1;
163 [COM] = reindex_com(COM);
164 COMTY.COM{1} = COM;
165 COMTY.MOD{1} = [Q{:}];
```

```
166 COMTY.Average(1) = Average;
167 COMTY.Variance(1) = Variance;
168 COMTY.Function(1) = Function;
169 COMTY.Niter(1) = Niter;
170
171 % Perform part 2
172 if (z == 1)
173   Mnew = M;
174   Mold = Mnew;
175   COMcur = COM;
176   COMfull = COM;
177   j = 2;
178
179   while 1
180     Mold = Mnew;
181     S2 = size(Mold);
182     Nnode = S2(1);
183
184     COMu = unique(COMcur);
185     Ncom = length(COMu);
186     ind_com = sparse(Ncom,Nnode);
187     ind_com_full = sparse(Ncom,N);
188     for p=1:Ncom
189       ind = find(COMcur==p);
190       ind_com(p,1:length(ind)) = ind;
191     end
192     for p=1:Ncom
193       ind = find(COMfull==p);
194       ind_com_full(p,1:length(ind)) = ind;
195     end
196     Mnew = [];
197     for s=1:k
198         Mnew(:,:,s) = zeros(Ncom,Ncom); %new matrix (each node  is a
              community)
199         for m=1:Ncom
200             for n=m:Ncom
201                 ind1 = ind_com(m,:);
202                 ind2 = ind_com(n,:);
203                 %weights of edges between communities
204                  Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)
                       ));
205                  Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)
                       ));
206             end
207         end
208     end
209     [COMt e] = community_variance_minus(Mnew,lambda,0);
210     if (e ~= 1)
211       COMfull = sparse(1,N);
```

```
212         COMcur = COMt.COM{1};
213         for p=1:Ncom
214           ind1 = ind_com_full(p,:);
215           COMfull(ind1(ind1>0)) = COMcur(p);
216         end
217         [COMfull2] = reindex_com(COMfull);
218         COMTY.COM{j} = COMfull2;
219         COMTY.MOD{j} = COMt.MOD{1};
220         COMTY.Average(j) = COMt.Average(1);
221         COMTY.Variance(j) = COMt.Variance(1);
222         COMTY.Function(j) = COMt.Function(1);
223         COMTY.Niter(j) = COMt.Niter;
224         Ind = (COMfull2 == COMTY.COM{j-1});
225         if (sum(Ind) == length(Ind))
226           return;
227         end
228       else
229         return;
230       end
231       j = j + 1;
232     end
233   end
234   end
235
236   % Re-index community partition by size
237   function [C Ss] = reindex_com(COMold)
238   C = sparse(1,length(COMold));
239   COMu = unique(COMold);
240   S = sparse(1,length(COMu));
241   for l=1:length(COMu)
242       S(l) = length(COMold(COMold==COMu(l)));
243   end
244   [Ss INDs] = sort(S,'descend');
245   for l=1:length(COMu)
246       C(COMold==COMu(INDs(l))) = l;
247   end
248   end
249   %Compute modulartiy
250   function MOD = compute_modularity(C,Mat)
251
252   S = size(Mat);
253   N = S(1);
254   m = sum(sum(Mat))/2; %total weight
255   MOD = 0;
256   COMu = unique(C);
257
258   for j=1:length(COMu)
259       Cj = (C==COMu(j)); %list of nodes in Cj
260       Ec = sum(sum(Mat(Cj,Cj))); %sum of weights between nodes in Cj
```

```
261       Et = sum(sum(Mat(Cj,:))); %sum of weights of nodes incident in nodes
              of Cj
262       if Et>0
263            MOD = MOD + Ec/(2*m)-(Et/(2*m))^2;
264       end
265  end
266  end
```

Appendix C

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

```
%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
```

```
29      z = 1;
30  end
31
32  S = size(M);
33  N = S(1);
34  if length(S)==3
35      k = S(3);
36  else k = 1;
37  end
38
39  ending = 0;
40
41  for s=1:k
42      M2(:,:,s)=M(:,:,s)-diag(diag(M(:,:,s)));
43  end
44
45  %total weight of the graph
46  for s=1:k
47      if z==1
48          M(:,:,s) = M(:,:,s) + diag(diag(M(:,:,s)));
49      end
50      m{s} = sum(sum(M(:,:,s)))/2;
51  end
52
53  Niter = 0; %number of iterations
54
55  if (sum(cellfun(@(x)x>0,m))==0) | N == 1
56    ending = 1;
57    COMTY = 0;
58    return;
59  end
60
61  %Delate layers with m=0
62  M(:,:,(cellfun(@(x)x==0,m)))=[];
63  k = k - sum(cellfun(@(x)x==0,m));
64
65  COM = 1:S(1); % Community of node i
66
67  for s=1:k
68      K(:,:,s) = sum(M(:,:,s)); % K(i)= sum of wieght incident to node i
69      SumTot(:,:,s) = sum(M(:,:,s)); %SumTot(i)= sum of weight incident to
            nodes in community i
70      SumIn(:,:,s) = diag(M(:,:,s))'; %SumIn(i)= sum of weight inside
          community i (loops)
71                                      %At the beginning each node is a
                                           community
72      Q{s} = compute_modularity(COM,M(:,:,s));
73  end
74  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;
    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(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
```

```
119          %variance gain
120          if k==1
121              GV_j = 0;
122          else
123              for s=1:k
124                  DQ{s} = GQ_i{s} + GQ_j{s}; %gains
125              end
126              M_DQ = sum([DQ{:}])/k; %gain average
127              GV_j = ((sum(([DQ{:}] - M_DQ).^(2))) / (k-1)) + (2/(k-1)) *
                     sum(sum(([Q{:}]-Average) .* ([DQ{:}] - M_DQ))); %gain
                     variance
128          end
129          %gain of the function
130          G(Cj) = (1-lambda)*M_DQ + lambda* GV_j;
131          if G(Cj) > best_increase
132            best_increase = G(Cj); %gain function
133            Q_t = DQ; %gain of modularity;
134            M_t = M_DQ; %gain average
135            V_t = GV_j;%gain of variance
136            Cnew_t = Cj;
137          end
138        end
139      end
140      if best_increase > -10^(-15)
141        Cnew = Cnew_t;
142        for s=1:k
143            Q{s} = Q{s} + Q_t{s};
144        end
145        Average = Average + M_t;
146        Variance = Variance + V_t;
147        Function = Function + best_increase;
148      end
149      %Recalculate
150      Ck = (COM==Cnew);
151      for s=1:k
152          SumIn(:,Cnew,s) = SumIn(:,Cnew,s) + 2*sum(M(i,Ck,s)) + M(i,i,s);
153          SumTot(:,Cnew,s) = SumTot(:,Cnew,s) + K(:,i,s);
154      end
155      COM(i) = Cnew;
156      if (Cnew ~= Ci)
157        gain = 1;
158      end
159    end
160    Niter = Niter + 1;
161  end
162  Niter = Niter - 1;
163  [COM] = reindex_com(COM);
164  COMTY.COM{1} = COM;
165  COMTY.MOD{1} = [Q{:}];
```

```
166 COMTY.Average(1) = Average;
167 COMTY.Variance(1) = Variance;
168 COMTY.Function(1) = Function;
169 COMTY.Niter(1) = Niter;
170
171 % Perform part 2
172 if (z == 1)
173   Mnew = M;
174   Mold = Mnew;
175   COMcur = COM;
176   COMfull = COM;
177   j = 2;
178
179   while 1
180     Mold = Mnew;
181     S2 = size(Mold);
182     Nnode = S2(1);
183
184     COMu = unique(COMcur);
185     Ncom = length(COMu);
186     ind_com = sparse(Ncom,Nnode);
187     ind_com_full = sparse(Ncom,N);
188     for p=1:Ncom
189       ind = find(COMcur==p);
190       ind_com(p,1:length(ind)) = ind;
191     end
192     for p=1:Ncom
193       ind = find(COMfull==p);
194       ind_com_full(p,1:length(ind)) = ind;
195     end
196     Mnew = [];
197     for s=1:k
198         Mnew(:,:,s) = zeros(Ncom,Ncom); %new matrix (each node  is a
              community)
199         for m=1:Ncom
200             for n=m:Ncom
201                 ind1 = ind_com(m,:);
202                 ind2 = ind_com(n,:);
203                 %weights of edges between communities
204                  Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)
                      ));
205                  Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0),s)
                      ));
206             end
207         end
208     end
209     [COMt e] = community_variance_plus(Mnew,lambda,0);
210     if (e ~= 1)
211       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.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;
        end
      else
        return;
      end
      j = j + 1;
    end
  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 modulartiy
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
```

```
261       Ec = sum(sum(Mat(Cj,Cj))); %sum of weights between nodes in Cj
262       Et = sum(sum(Mat(Cj,:))); %sum of weights of nodes incident in nodes
              of Cj
263       if Et>0
264           MOD = MOD + Ec/(2*m)-(Et/(2*m))^2;
265       end
266   end
267   end
```

Appendix D

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

```
%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;
```

```
30 end
31
32 S = size(M);
33 N = S(1);
34 if length(S)==3
35     k = S(3);
36 else k = 1;
37 end
38
39 ending = 0;
40
41 for s=1:k
42     M2(:,:,s)=M(:,:,s)-diag(diag(M(:,:,s)));
43 end
44
45 %total weight of the graph
46 for s=1:k
47     if z==1
48         M(:,:,s) = M(:,:,s) + diag(diag(M(:,:,s)));
49     end
50     m{s} = sum(sum(M(:,:,s)))/2;
51 end
52
53 Niter = 0; %number of iterations
54
55 % Calculation of the beginning values
56 COM{1} = 1:S(1); %current community %At the beginning each node is a
       community
57 COM{2} = 1:S(1); %original graph commuity
58 COM{3} = 1:S(1); %community reindexed
59 %COM(i)= Community of node i
60
61 for s=1:k
62     K(:,:,s) = sum(M(:,:,s)); % K(i)= sum of wieght incident to node i
63     SumTot(:,:,s) = sum(M(:,:,s)); %SumTot(i)= sum of weight incident to
           nodes in community i
64     SumIn(:,:,s) = diag(M(:,:,s))'; %SumIn(i)= sum of weight inside
          community i (loops)
65                                     %At the beginning each node is a
                                          community
66     Q{s} = compute_modularity(COM{1},M(:,:,s));
67 end
68 %average
69 Average =  sum([Q{:}]) / k;
70
71 %filter
72 L={[{Q},{COM},SumTot,SumIn,Average]};
73 %If no edges in any layer or just one node
74 if (sum(cellfun(@(x)x>0,m))==0) | N == 1 %| logical or
```

```
75    ending = 1;
76  else
77
78  %Delate layers with m=0
79  M(:,:,(cellfun(@(x)x==0,m)))=[];
80  k = k - sum(cellfun(@(x)x==0,m));
81
82  %Neighbor{j}{s} neighbor of node j in layer s
83  for j=1:N
84      for s=1:k
85          temp=M2(:,:,s);
86          Neighbor_j{s} = find(temp(j,:));
87      end
88      Neighbor{j}=Neighbor_j;
89  end
90
91
92  GAIN = 1;
93  while (GAIN == 1) %Stop when putting nodes in other communities do not
        increase the modularity
94    L_old=L;
95    L_new=L;
96    for i=1:N %for each node, in this order
97        L_o=L;
98        for l=1:length(L_o) %for each situation in the filter
99            %delate the point from the filter
100           L_new_o=L_new;
101           index = cellfun(@(x) isequal(x,L_o{l}), L_new, 'UniformOutput'
                  , 1);
102           L_new(index)=[];
103           %step1
104           [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,M,N,m,L_new,h,i);
105           %if no new point, insert again the initial point in the filter
106           if U==0
107             L_new=L_new_o;
108           end
109       end
110       L_new = cut_filter(L_new,h,k);
111       L=L_new;
112   end
113   %Check if nothing happened
114   if length(L_old)~=length(L_new)
115       GAIN=1;
116   else
117       C_old=[];
118       C_new=[];
119       for l=1:length(L_old)
120           R_old=[];
```

```
121             R_new=[];
122             for s=1:k
123                 R_old = [R_old L_old{l}{1}{s}];
124                 R_new = [R_new L_new{l}{1}{s}];
125             end
126             C_old = [C_old ; R_old];
127             C_new = [C_new ; R_new];
128         end
129         C = ismembertol(C_old,C_new,10^(-6));
130         if size(C,1)*size(C,2) == sum(sum(C))
131             GAIN=0;
132         end
133     end
134     Niter = Niter + 1;
135 end
136 end
137
138 % Perform part 2
139 if (z==1)
140         L = cut_filter(L,1,k); %one final element
141         Mnew=M; %new weight matrix of this iteration
142         Mold=Mnew; %old weight matrix of iteration befor
143         COMcur = L{1}{2}{3}; %communities in the graph of this iteration
144         COMfull = L{1}{2}{3}; %communities in the original graph
145         LL=L;
146         j=2;%number of pass (step1+step2)
147         S=1;
148         while S
149           L_old=LL;
150           Mold = Mnew;
151           S2 = size(Mold); %number of nodes
152           Nnode = S2(1);
153
154           COMu = unique(COMcur); %list of communities without repetitions
                 and in sorted order
155           Ncom = length(COMu); %number of communities
156           ind_com = sparse(Ncom,Nnode); %zero matrix with a row for each
                 community and a column for each node in this configuration
157           ind_com_full =sparse(Ncom,N); %zero matrix with a row for each
                 community and a column for each node of the original graph
158
159           for p=1:Ncom %for each community
160               ind = find(COMcur==p);
161               ind_com(p,1:length(ind)) = ind;
162           end
163           for p=1:Ncom
164               ind = find(COMfull==p);
165               ind_com_full(p,1:length(ind)) = ind;
166           end
```

```
167         Mnew=[];
168         for s=1:k
169             Mnew(:,:,s) = sparse(Ncom,Ncom); %new matrix (each node  is
                a community)
170             for m=1:Ncom
171                 for n=m:Ncom
172                     ind1 = ind_com(m,:);
173                     ind2 = ind_com(n,:);
174                     %weights of edges between communities
175                     Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0)
                        ,s)));
176                     Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0)
                        ,s)));
177                 end
178             end
179         end
180         %apply first step to this new matrix but z=0 without recursive
181         [LLL e] = multi_average(Mnew,h,0);
182         if isempty(LLL)
183             L=LLL;
184             return
185         else
186         LL = cut_filter(LLL,1,k); %one final element
187         %if (e ~= 1)
188             COMfull = sparse(1,N); %communities of the original graph
189             COMcur = LL{1}{2}{3}; %communities
190             for p=1:Ncom
191                 ind1 = ind_com_full(p,:); %nodes in community p in the
                    original graph
192                 COMfull(ind1(ind1>0)) = COMcur(p); %community now of
                    node i
193             end
194             [COMfull] = reindex_com(COMfull);
195             LL{1}{2}{2}=COMfull;
196             %communities do not change
197             if isequal(L_old{1}{2}{2},LL{1}{2}{2})
198                 %return not just this value, but all the value of the
                    last list
199                 %calculate COMfull for each element of the last list
200                 for l=1:length(LLL)
201                     COMfull = sparse(1,N); %communities of the original
                        graph
202                     COMcur = LLL{l}{2}{3}; %communities
203                     for p=1:Ncom
204                         ind1 = ind_com_full(p,:); %nodes in community p
                            in the original graph
205                         COMfull(ind1(ind1>0)) = COMcur(p); %community
                            now of node i
206                     end
```

```
                    [COMfull] = reindex_com(COMfull);
                    LLL{l}{2}{2}=COMfull;
                end
                L = LLL;
                S=0;
            end
        j = j + 1; %start another pass
        tEnd = toc;
        if tEnd > 1200
            L = {};
            return
        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;

%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)));
    %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;
```

```
251     SumIn3=SumIn2;
252     Cj = COMcur(NB(j)); %community of node j
253     if (G(Cj) == 0) %If not tried with another node of community Cj yet
254         G(Cj)=1;
255         for s=1:k
256             COMcur3 = COMcur;
257             COMcur3(i) = -1;
258             %I put i in the community of j for each layer
259             CNj = (COMcur3==Cj); %nodes in community Cj, without j
260             Ki_in_j{s} = sum(M(i,CNj,s)); %sum of weights between i and
                    Cj
261             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
262             %Recalculate
263             SumTot3(:,Cj,s) = SumTot3(:,Cj,s) + K(:,i,s);
264             SumIn3(:,Cj,s) = SumIn3(:,Cj,s) + 2*sum(M(i,CNj,s)) + M(i,i,
                    s);
265         end
266     COMcur4=COMcur;
267     COMcur4(i)=Cj;
268     COM{1}{1}=COMcur4;
269     [COMcur4] = reindex_com(COMcur4);
270     COM{1}{3}=COMcur4;
271     for s=1:k
272         DQ{s} = GQ_i{s} + GQ_j{s}; %gain modularity
273     end
274     GA_j = sum([DQ{:}])/k; %gain average
275     if GA_j > -10^(-14)
276         A_j = Average + GA_j;
277         %modularity
278         for s=1:k
279             Q_j{s} = Q{1}{s} + DQ{s};
280         end
281         [L] = add_to_filter({Q_j},COM,SumTot3,SumIn3,A_j,L,k);
282         U=1;
283     end
284     end
285 end
286 end
287
288 %add a point to the filter
289 function [L] = add_to_filter(QQ,COM,SumTot,SumIn,Average,L,k)
290
291 if length(L) ~= 0
292
293 COMcur=COM{1}{1}; %current community
294
295 %1. Check if the new point is dominated by a point in the filter
296 DD=0;
```

```
297 l=1;
298 while (DD==0) & l<=length(L)
299         D=0;
300         s=1;
301         while (D==0) & s<=k
302                 if QQ{1}{s}>L{l}{1}{s}
303                     D=1;
304                 end
305                 s=s+1;
306         end
307         if D %the new point is not dominated by l
308             DD=1;
309         end
310         l=l+1;
311 end
312 if  DD %the new point is not dominated
313   DDD=0;
314   l=1;
315   while (DDD==0) & l<=length(L)
316       D=1;
317       s=1;
318       while D & s<=k
319           if abs(L{l}{1}{s} - QQ{1}{s}) > 10^(-14)
320               D=0;
321           end
322           s = s + 1;
323       end
324       if D
325           DDD=1;
326       end
327       l = l + 1;
328   end
329
330   if DDD==0
331
332   %2.Check if the other points in the filter are dominated by the new
       point %
333
334   for l=1:length(L)
335         D=1;
336         s=1;
337         while D & s<=k
338                 if L{l}{1}{s}>QQ{1}{s}+10^(-14)
339                     D=0;
340                 end
341                 s=s+1;
342         end
343         if D %l is dominated by the new point so I remove it from the
              filter
```

```
344                 L{l}=[];
345             end
346     end
347
348     empties =(cellfun(@isempty,L));
349     L(empties) = [];
350     L{end+1}=[QQ,COM,SumTot,SumIn,Average];  %Add the new point to the
          filter
351
352 end
353 end
354 else
355         L{end+1}=[QQ,COM,SumTot,SumIn,Average];  %Add the new point to the
              filter
356 end
357 end
358
359 %cut the filter
360 function [L] = cut_filter(L,h,k)
361
362 %calculate the function for every element of the list
363 if length(L)>h
364     for l=1:length(L)
365         %function
366         Average{l} = L{l}{5};
367     end
368
369 %remuve the situations in the filter with the lower values
370 for t=1:(length(L)-h)
371     [mn,idx]=min([Average{:}]);
372     v=find([Average{:}]==mn); %if more element with same average values,
          delate the last one (because with h=1 I have same result as
          cluster_jl_average_variance)
373     idx_t=v(end);
374     Average{idx_t}=Inf;
375     L{idx_t}=[];
376 end
377
378 empties = find(cellfun(@isempty,L));
379 L(empties) = [];
380 end
381 end
382
383 %Compute modulartiy
384 function MOD = compute_modularity(C,Mat)
385
386 S = size(Mat);
387 N = S(1);
388
```

```
389 m = sum(sum(Mat))/2; %total weight
390
391 MOD = 0;
392 COMu = unique(C); %list of communities without repetiotions and in
        sorted order
393 %for each community calculate modularity and then sum all together
394 for j=1:length(COMu)
395     Cj = (C==COMu(j)); %list of nodes in Cj
396                        %faster then Cj = find(C==COMu(j))
397     Ec = sum(sum(Mat(Cj,Cj))); %sum of weights between nodes in Cj
398                                %Mat(Cj,Cj) submatrix
399     Et = sum(sum(Mat(Cj,:))); %sum of weights of nodes incident in nodes
            of Cj
400     if Et>0
401         MOD = MOD + Ec/(2*m)-(Et/(2*m))^2;
402     end
403 end
404
405 end
406
407 % Re-index community partition by size
408 function [C Ss] = reindex_com(COMold)
409 C = sparse(1,length(COMold));
410 COMu = unique(COMold);
411 S = sparse(1,length(COMu));
412 for l=1:length(COMu)
413     S(l) = length(COMold(COMold==COMu(l)));
414 end
415 [Ss INDs] = sort(S,'descend');
416 for l=1:length(COMu)
417     C(COMold==COMu(INDs(l))) = l;
418 end
419 end
420 %Re-index community partition no by size but by initial order
421 %function [C] = reindex_com(COMold)
422 %C = sparse(1,length(COMold)); %vector with a index for each node
423 %COMu = unique(COMold); %unique(v)=same data of v but not repetitions
        and in sorted order
424 %COMolds=sort(COMold);
425 %COMu=COMolds([true;diff(COMolds(:))>0]);
426 %for l=1:length(COMu)
427 %    C(COMold==COMu(l)) = l; %Riname the communities
428 %end
429 %end
```

Appendix E

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

```
%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');
```

```
27 end
28 if nargin < 3
29   error('not h defined');
30 end
31 if nargin < 4
32   z = 1;
33 end
34
35 S = size(M);
36 N = S(1);
37 if length(S)==3
38     k = S(3);
39 else k = 1;
40 end
41
42 ending = 0;
43
44 for s=1:k
45     M2(:,:,s)=M(:,:,s)-diag(diag(M(:,:,s)));
46 end
47
48 %total weight of the graph
49 for s=1:k
50     if z==1
51         M(:,:,s) = M(:,:,s) + diag(diag(M(:,:,s)));
52     end
53     m{s} = sum(sum(M(:,:,s)))/2;
54 end
55
56 Niter = 0; %number of iterations
57
58 % Calculation of the beginning values
59 COM{1} = 1:S(1); %current community %At the beginning each node is a
       community
60 COM{2} = 1:S(1); %original graph commuity
61 COM{3} = 1:S(1); %community reindexed
62 %COM(i)= Community of node i
63
64 for s=1:k
65     K(:,:,s) = sum(M(:,:,s)); % K(i)= sum of wieght incident to node i
66     SumTot(:,:,s) = sum(M(:,:,s)); %SumTot(i)= sum of weight incident to
           nodes in community i
67     SumIn(:,:,s) = diag(M(:,:,s))'; %SumIn(i)= sum of weight inside
          community i (loops)
68                                       %At the beginning each node is a
                                             community
69     Q{s} = compute_modularity(COM{1},M(:,:,s));
70 end
71 %average
```

```
72  Average =  sum([Q{:}]) / k;
73  %variance
74  if k==1
75     Variance = 0;
76  else
77     Variance = (sum(([Q{:}]-Average).^(2))) / (k-1);
78  end
79  %function
80  Function = (1-lambda) * Average - lambda * Variance;
81
82  %filter
83  L={[{Q},{COM},SumTot,SumIn,Average,Function]};
84  %If no edges in any layer or just one node
85  if (sum(cellfun(@(x)x>0,m))==0) | N == 1 %| logical or
86    ending = 1;
87  else
88
89  %Delate layers with m=0
90  M(:,:,(cellfun(@(x)x==0,m)))=[];
91  k = k - sum(cellfun(@(x)x==0,m));
92
93  %Neighbor{j}{s} neighbor of node j in layer s
94  for j=1:N
95      for s=1:k
96          temp=M2(:,:,s);
97          Neighbor_j{s} = find(temp(j,:));
98      end
99      Neighbor{j}=Neighbor_j;
100 end
101
102
103 GAIN = 1;
104 while (GAIN == 1) %Stop when putting nodes in other communities do not
        increase the modularity
105   L_old=L;
106   L_new=L;
107   for i=1:N %for each node, in this order
108       L_o=L;
109       for l=1:length(L_o) %for each situation in the filter
110           %delate the point from the filter
111           L_new_o=L_new;
112           index = cellfun(@(x) isequal(x,L_o{l}), L_new, 'UniformOutput'
                  , 1);
113           L_new(index)=[];
114           %step1
115           [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);
116           %if no new point, put again the initial point in the filter
```

```
117             if U==0
118               L_new=L_new_o;
119             end
120         end
121         L_new = cut_filter(L_new,h,k,lambda);
122         L=L_new;
123     end
124     %Check if nothing happened
125     if length(L_old)~=length(L_new)
126         GAIN=1;
127     else
128         C_old=[];
129         C_new=[];
130         for l=1:length(L_old)
131             R_old=[];
132             R_new=[];
133             for s=1:k
134                 R_old = [R_old L_old{l}{1}{s}];
135                 R_new = [R_new L_new{l}{1}{s}];
136             end
137             C_old = [C_old ; R_old];
138             C_new = [C_new ; R_new];
139         end
140         C = ismembertol(C_old,C_new,10^(-6));
141         if size(C,1)*size(C,2) == sum(sum(C))
142             GAIN=0;
143         end
144     end
145     Niter = Niter + 1;
146 end
147 end
148
149 % Perform part 2
150 if (z==1)
151         L = cut_filter(L,1,k); %one final element
152         Mnew=M; %new weight matrix of this iteration
153         Mold=Mnew; %old weight matrix of iteration befor
154         COMcur = L{1}{2}{3}; %communities in the graph of this iteration
155         COMfull = L{1}{2}{3}; %communities in the original graph
156         LL=L;
157         j=2;%number of pass (step1+step2)
158         S=1;
159         while S
160           L_old=LL;
161           Mold = Mnew;
162           S2 = size(Mold); %number of nodes
163           Nnode = S2(1);
164
165           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 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);
```

```
205             LL{1}{2}{2}=COMfull;
206             %communities do not change
207             if isequal(L_old{1}{2}{2},LL{1}{2}{2})
208                 %return not just this value, but all the value of the
                        last list
209                 %calculate COMfull for each element of the last list
210                 for l=1:length(LLL)
211                     COMfull = sparse(1,N); %communities of the original
                            graph
212                     COMcur = LLL{l}{2}{3}; %communities
213                     for p=1:Ncom
214                         ind1 = ind_com_full(p,:); %nodes in community p
                                in the original graph
215                         COMfull(ind1(ind1>0)) = COMcur(p); %community
                                now of node i
216                     end
217                     [COMfull] = reindex_com(COMfull);
218                     LLL{l}{2}{2}=COMfull;
219                 end
220                 L = LLL;
221                 S=0;
222             end
223         j = j + 1; %start another pass
224         tEnd = toc;
225         if tEnd > 1200
226             L = {};
227             return
228         end
229         end
230       end
231 end
232 end
233
234 %Compute step_1
235 function [L,U] = step_1(Q,COM,SumTot,SumIn,Average,Function,k,Neighbor,K
        ,M,N,m,L,h,lambda,i)
236
237 U=0; %if U=1 add at least an element to the list, otherwise insert again
         the initial point in the filter
238 COMcur=COM{1}{1}; %current community
239 Ci = COMcur(i); %community of node i
240
241 NB=unique(cat(2,Neighbor{i}{:}));
242
243 SumTot2=SumTot;
244 SumIn2=SumIn;
245
246 %remove i from its community
247 for s=1:k
```

```
248         COMcur2 = COMcur;
249         COMcur2(i) = -1;
250         CNi = (COMcur2==Ci); %list of nodes in Ci community, without i
251         Ki_in_i{s} = sum(M(i,CNi,s)); %sum of weights between i and Ci
252         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)));
253         %Recalculate values
254         SumTot2(:,Ci,s) = SumTot2(:,Ci,s) - K(:,i,s); %weights incident to
                Ci community
255         SumIn2(:,Ci,s) = SumIn2(:,Ci,s) - 2*sum(M(i,CNi,s)) - M(i,i,s); %
                weights community i
256     end
257
258     G = sparse(1,N);
259     for j=1:length(NB)
260         SumTot3=SumTot2;
261         SumIn3=SumIn2;
262         Cj = COMcur(NB(j)); %community of node j
263         if (G(Cj) == 0) %If not tried with another node of community Cj yet
264             G(Cj)=1;
265             for s=1:k
266                 COMcur3 = COMcur;
267                 COMcur3(i) = -1;
268                 %Insert i in the community of j for each layer
269                 CNj = (COMcur3==Cj); %nodes in community Cj, without j
270                 Ki_in_j{s} = sum(M(i,CNj,s)); %sum of weights between i and
                        Cj
271                 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
272                 %Recalculate
273                 SumTot3(:,Cj,s) = SumTot3(:,Cj,s) + K(:,i,s);
274                 SumIn3(:,Cj,s) = SumIn3(:,Cj,s) + 2*sum(M(i,CNj,s)) + M(i,i,
                        s);
275             end
276         COMcur4=COMcur;
277         COMcur4(i)=Cj;
278         COM{1}{1}=COMcur4;
279         [COMcur4] = reindex_com(COMcur4);
280         COM{1}{3}=COMcur4;
281
282         %variance
283         if k==1
284             GV_j = 0;
285         else
286             for s=1:k
287                 DQ{s} = GQ_i{s} + GQ_j{s}; %gains
288             end
289             M_DQ = sum([DQ{:}])/k; %gain average
290             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

%1. 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
```

```
339             if abs(L{l}{1}{s} - QQ{1}{s}) > 10^(-14)
340                 D=0;
341             end
342             s = s + 1;
343         end
344         if D
345             DDD=1;
346         end
347         l = l + 1;
348     end
349
350     if DDD==0
351
352     %2.Check if the other points in the filter are dominated by the new
        point %
353
354     for l=1:length(L)
355           D=1;
356           s=1;
357           while D & s<=k
358                   if L{l}{1}{s}>QQ{1}{s}+10^(-14)
359                       D=0;
360                   end
361                   s=s+1;
362           end
363           if D %l is dominated by the new point so I remove it from the
              filter
364               L{l}=[];
365           end
366     end
367
368     empties =(cellfun(@isempty,L));
369     L(empties) = [];
370     L{end+1}=[QQ,COM,SumTot,SumIn,Average,Function];  %I add the new point
          to the filter
371
372 end
373 end
374 else
375     L{end+1}=[QQ,COM,SumTot,SumIn,Average,Function];  %I add the new
          point to the filter
376 end
377 end
378
379 %cut the filter
380 function [L] = cut_filter(L,h,k,lambda)
381
382 %calculate the function for every element of the list
383 if length(L)>h
```

```
384     for l=1:length(L)
385         %function
386         Function{l} = L{l}{6};
387     end
388 
389 %remuve the situations in the filter with the lower values
390 for t=1:(length(L)-h)
391   [mn,idx]=min([Function{:}]);
392   v=find([Function{:}]==mn); %if more element with same average values,
        delate the last one (because with h=1 I have same result as
        cluster_jl_average_variance)
393   idx_t=v(end);
394   Function{idx_t}=Inf;
395   L{idx_t}=[];
396 end
397 
398 empties = find(cellfun(@isempty,L));
399 L(empties) = [];
400 end
401 end
402 
403 %Compute modulartiy
404 function MOD = compute_modularity(C,Mat)
405 
406 S = size(Mat);
407 N = S(1);
408 
409 m = sum(sum(Mat))/2; %total weight
410 
411 MOD = 0;
412 COMu = unique(C); %list of communities without repetiotions and in
    sorted order
413 %for each community calculate modularity and then sum all together
414 for j=1:length(COMu)
415     Cj = (C==COMu(j)); %list of nodes in Cj
416     Ec = sum(sum(Mat(Cj,Cj))); %sum of weights between nodes in Cj
417     Et = sum(sum(Mat(Cj,:))); %sum of weights of nodes incident in nodes
          of Cj
418     if Et>0
419         MOD = MOD + Ec/(2*m)-(Et/(2*m))^2;
420     end
421 end
422 
423 end
424 
425 % Re-index community partition by size
426 function [C Ss] = reindex_com(COMold)
427 C = sparse(1,length(COMold));
428 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)); %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; %Riname 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  (funtion)
%


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

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

```
27 if nargin < 2
28   error('not lambda defined');
29 end
30 if nargin < 3
31   error('not h defined');
32 end
33 if nargin < 4
34   z = 1;
35 end
36
37 S = size(M);
38 N = S(1);
39 if length(S)==3
40     k = S(3);
41 else k = 1;
42 end
43
44 ending = 0;
45
46 for s=1:k
47     M2(:,:,s)=M(:,:,s)-diag(diag(M(:,:,s)));
48 end
49
50 %total weight of the graph
51 for s=1:k
52     if z==1
53         M(:,:,s) = M(:,:,s) + diag(diag(M(:,:,s)));
54     end
55     m{s} = sum(sum(M(:,:,s)))/2;
56 end
57
58 Niter = 0; %number of iterations
59
60 % Calculation of the beginning values
61 COM{1} = 1:S(1); %current community %At the beginning each node is a
      community
62 COM{2} = 1:S(1); %original graph commuity
63 COM{3} = 1:S(1); %community reindexed
64 %COM(i)= Community of node i
65
66 for s=1:k
67     K(:,:,s) = sum(M(:,:,s)); % K(i)= sum of wieght incident to node i
68     SumTot(:,:,s) = sum(M(:,:,s)); %SumTot(i)= sum of weight incident to
          nodes in community i
69     SumIn(:,:,s) = diag(M(:,:,s))'; %SumIn(i)= sum of weight inside
         community i (loops)
70                                     %At the beginning each node is a
                                           community
71     Q{s} = compute_modularity(COM{1},M(:,:,s));
```

```
72  end
73  %average
74  Average =  sum([Q{:}]) / k;
75  %variance
76  if k==1
77     Variance = 0;
78  else
79     Variance = (sum(([Q{:}]-Average).^(2))) / (k-1);
80  end
81  %function
82  Function = (1-lambda) * Average + lambda * Variance;
83
84  %filter
85  L={[{Q},{COM},SumTot,SumIn,Average,Function]};
86  %If no edges in any layer or just one node
87  if (sum(cellfun(@(x)x>0,m))==0) | N == 1 %| logical or
88    ending = 1;
89  else
90
91  %Delate layers with m=0
92  M(:,:,(cellfun(@(x)x==0,m)))=[];
93  k = k - sum(cellfun(@(x)x==0,m));
94
95  %Neighbor{j}{s} neighbor of node j in layer s
96  for j=1:N
97      for s=1:k
98          temp=M2(:,:,s);
99          Neighbor_j{s} = find(temp(j,:));
100     end
101     Neighbor{j}=Neighbor_j;
102 end
103
104
105 GAIN = 1;
106 while (GAIN == 1) %Stop when putting nodes in other communities do not
       increase the modularity
107 %while Niter<=2
108   L_old=L;
109   L_new=L;
110   for i=1:N %for each node, in this order
111       L_o=L;
112       for l=1:length(L_o) %for each situation in the filter
113           %delate the point from the filter
114           L_new_o=L_new;
115           index = cellfun(@(x) isequal(x,L_o{l}), L_new, 'UniformOutput'
                , 1);
116           L_new(index)=[];
117           %step1
118           [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);
119             %if no new point, insert again the initial point in the filter
120             if U==0
121               L_new=L_new_o;
122             end
123         end
124         L_new = cut_filter(L_new,h,k,lambda);
125         L=L_new;
126     end
127     %Check if nothing happened
128     if length(L_old)~=length(L_new)
129         GAIN=1;
130     else
131         C_old=[];
132         C_new=[];
133         for l=1:length(L_old)
134             R_old=[];
135             R_new=[];
136             for s=1:k
137                 R_old = [R_old L_old{l}{1}{s}];
138                 R_new = [R_new L_new{l}{1}{s}];
139             end
140             C_old = [C_old ; R_old];
141             C_new = [C_new ; R_new];
142         end
143         C = ismembertol(C_old,C_new,10^(-6));
144         if size(C,1)*size(C,2) == sum(sum(C))
145             GAIN=0;
146         end
147     end
148     Niter = Niter + 1;
149     tEnd = toc;
150     if tEnd > 500
151        L={};
152        return
153     end
154 end
155 end
156
157 % Perform part 2
158 if (z==1)
159         L = cut_filter(L,1,k); %one final element
160         Mnew=M; %new weight matrix of this iteration
161         Mold=Mnew; %old weight matrix of iteration befor
162         COMcur = L{1}{2}{3}; %communities in the graph of this iteration
163         COMfull = L{1}{2}{3}; %communities in the original graph
164         LL=L;
165         j=2;%number of pass (step1+step2)
```

```
166         S=1;
167         while S
168           L_old=LL;
169           Mold = Mnew;
170           S2 = size(Mold); %number of nodes
171           Nnode = S2(1);
172 
173           COMu = unique(COMcur); %list of communities without repetitions
                 and in sorted order
174           Ncom = length(COMu); %number of communities
175           ind_com = sparse(Ncom,Nnode); %zero matrix with a row for each
                 community and a column for each node in this configuration
176           ind_com_full =sparse(Ncom,N); %zero matrix with a row for each
                 community and a column for each node of the original graph
177 
178           for p=1:Ncom %for each community
179               ind = find(COMcur==p);
180               ind_com(p,1:length(ind)) = ind; %in row p insert the indeces
                      of the nodes in community p in this configuration
181           end
182           for p=1:Ncom
183               ind = find(COMfull==p);
184               ind_com_full(p,1:length(ind)) = ind; %in row p insert the
                     indeces of the nodes in community p in the original graph
185           end
186           Mnew=[];
187           for s=1:k
188               Mnew(:,:,s) = sparse(Ncom,Ncom); %new matrix (each node  is
                     a community)
189               for m=1:Ncom
190                   for n=m:Ncom
191                       ind1 = ind_com(m,:);
192                       ind2 = ind_com(n,:);
193                       %weights of edges between communities
194                       Mnew(m,n,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0)
                             ,s)));
195                       Mnew(n,m,s) = sum(sum(Mold(ind1(ind1>0),ind2(ind2>0)
                             ,s)));
196                   end
197               end
198           end
199           %apply first step to this new matrix but z=0 without recursive
200           [LLL e] = multi_variance_plus(Mnew,lambda,h,0);
201           if isempty(LLL)
202               L=LLL;
203               return
204           else
205           LL = cut_filter(LLL,1,k); %one final element
206               COMfull = sparse(1,N); %communities of the original graph
```

```
207             COMcur = LL{1}{2}{3}; %communities
208             for p=1:Ncom
209                 ind1 = ind_com_full(p,:); %nodes in community p in the
                        original graph
210                 COMfull(ind1(ind1>0)) = COMcur(p); %community now of
                        node i
211             end
212             [COMfull] = reindex_com(COMfull);
213             LL{1}{2}{2}=COMfull;
214             %communities do not change
215             if isequal(L_old{1}{2}{2},LL{1}{2}{2})
216                 %return not just this value, but all the value of the
                        last list
217                 %calculate COMfull for each element of the last list
218                 for l=1:length(LLL)
219                     COMfull = sparse(1,N); %communities of the original
                            graph
220                     COMcur = LLL{l}{2}{3}; %communities
221                     for p=1:Ncom
222                         ind1 = ind_com_full(p,:); %nodes in community p
                                in the original graph
223                         COMfull(ind1(ind1>0)) = COMcur(p); %community
                                now of node i
224                     end
225                     [COMfull] = reindex_com(COMfull);
226                     LLL{l}{2}{2}=COMfull;
227                 end
228                 L = LLL;
229                 S=0;
230             end
231         j = j + 1; %start another pass
232         tEnd = toc;
233         if tEnd > 600
234             L = {};
235             return
236         end
237         end
238       end
239 end
240 end
241
242 %Compute step_1
243 function [L,U] = step_1(Q,COM,SumTot,SumIn,Average,Function,k,Neighbor,K
        ,M,N,m,L,h,lambda,i)
244
245 U=0; %if U=1 add at least an element to the list, otherwise insert again
        the initial point in the filter
246 COMcur=COM{1}{1}; %current community
247 Ci = COMcur(i); %community of node i
```

```
248
249 NB=unique(cat(2,Neighbor{i}{:}));
250
251 SumTot2=SumTot;
252 SumIn2=SumIn;
253
254 %remove i from its community
255 for s=1:k
256     COMcur2 = COMcur;
257     COMcur2(i) = -1;
258     CNi = (COMcur2==Ci); %list of nodes in Ci community, without i
259     Ki_in_i{s} = sum(M(i,CNi,s)); %sum of weights between i and Ci
260     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)));
261     %Recalculate values
262     SumTot2(:,Ci,s) = SumTot2(:,Ci,s) - K(:,i,s); %weights incident to
        Ci community
263     SumIn2(:,Ci,s) = SumIn2(:,Ci,s) - 2*sum(M(i,CNi,s)) - M(i,i,s); %
        weights community i
264 end
265
266 G = sparse(1,N);
267 for j=1:length(NB)
268     SumTot3=SumTot2;
269     SumIn3=SumIn2;
270     Cj = COMcur(NB(j)); %community of node j
271     if (G(Cj) == 0) %If have not tried with another node of community Cj
         yet
272         G(Cj)=1;
273         for s=1:k
274             COMcur3 = COMcur;
275             COMcur3(i) = -1;
276             %I put i in the community of j for each layer
277             CNj = (COMcur3==Cj); %nodes in community Cj, without j
278                                  %faster then CNj = find(COMcur==Cj)
279             Ki_in_j{s} = sum(M(i,CNj,s)); %sum of weights between i and
                Cj
280             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
281             %Recalculate
282             SumTot3(:,Cj,s) = SumTot3(:,Cj,s) + K(:,i,s);
283             SumIn3(:,Cj,s) = SumIn3(:,Cj,s) + 2*sum(M(i,CNj,s)) + M(i,i,
                s);
284         end
285     COMcur4=COMcur;
286     COMcur4(i)=Cj;
287     COM{1}{1}=COMcur4;
288     [COMcur4] = reindex_com(COMcur4);
289     COM{1}{3}=COMcur4;
```

```
290
291      %variance
292      if k==1
293         GV_j = 0;
294      else
295         for s=1:k
296            DQ{s} = GQ_i{s} + GQ_j{s}; %gains
297         end
298          M_DQ = sum([DQ{:}])/k; %gain average
299          GV_j = ((sum(([DQ{:}] - M_DQ).^(2))) / (k-1)) + (2/(k-1)) * sum(
                sum(([Q{1}{:}]-Average) .* ([DQ{:}] - M_DQ))); %gain variance
300      end
301      GF_j = (1-lambda)*M_DQ + lambda* GV_j;
302      if GF_j > -10^(-14)
303          F_j = Function + GF_j;
304          %modularity
305          for s=1:k
306              Q_j{s} = Q{1}{s} + DQ{s};
307          end
308          %average
309          A_j = Average + M_DQ;
310          [L] = add_to_filter({Q_j},COM,SumTot3,SumIn3,A_j,F_j,L,k);
311          U=1;
312      end
313      end
314  end
315  end
316
317  %add a point to the filter
318  function [L] = add_to_filter(QQ,COM,SumTot,SumIn,Average,Function,L,k)
319
320  if length(L) ~= 0
321
322  COMcur=COM{1}{1}; %current community
323
324  %1. Check if the new point is dominated by a point in the filter
325  DD=0;
326  l=1;
327  while (DD==0) & l<=length(L)
328          D=0;
329          s=1;
330          while (D==0) & s<=k
331                  if QQ{1}{s}>L{l}{1}{s}
332                      D=1;
333                  end
334                  s=s+1;
335          end
336          if D %the new point is not dominated by l
337              DD=1;
```

```
338          end
339          l=l+1;
340  end
341  if  DD %the new point is not dominated
342    DDD=0;
343    l=1;
344    while (DDD==0) & l<=length(L)
345        D=1;
346        s=1;
347        while D & s<=k
348            if abs(L{l}{1}{s} - QQ{1}{s}) > 10^(-14)
349                D=0;
350            end
351            s = s + 1;
352        end
353        if D
354            DDD=1;
355        end
356        l = l + 1;
357    end
358
359    if DDD==0
360
361    %2.Check if the other points in the filter are dominated by the new
       point %
362
363    for l=1:length(L)
364          D=1;
365          s=1;
366          while D & s<=k
367                  if L{l}{1}{s}>QQ{1}{s}+10^(-14)
368                      D=0;
369                  end
370                  s=s+1;
371          end
372          if D %l is dominated by the new point so I remove it from the
                filter
373              L{l}=[];
374          end
375    end
376
377    empties =(cellfun(@isempty,L));
378    L(empties) = [];
379    L{end+1}=[QQ,COM,SumTot,SumIn,Average,Function];  %I add the new point
       to the filter
380
381  end
382  end
383  else
```

```
384      L{end+1}=[QQ,COM,SumTot,SumIn,Average,Function];  %I add the new
             point to the filter
385  end
386  end
387
388  %cut the filter
389  function [L] = cut_filter(L,h,k,lambda)
390
391  %calculate the function for every element of the list
392  if length(L)>h
393     for l=1:length(L)
394          %function
395          Function{l} = L{l}{6};
396     end
397
398  %remuve the situations in the filter with the lower values
399  for t=1:(length(L)-h)
400    [mn,idx]=min([Function{:}]);
401    v=find([Function{:}]==mn); %if more element with same average values,
         delate the last one (because with h=1 I have same result as
         cluster_jl_average_variance)
402    idx_t=v(end);
403    Function{idx_t}=Inf;
404    L{idx_t}=[];
405  end
406
407  empties = find(cellfun(@isempty,L));
408  L(empties) = [];
409  end
410  end
411
412  %Compute modulartiy
413  function MOD = compute_modularity(C,Mat)
414
415  S = size(Mat);
416  N = S(1);
417
418  m = sum(sum(Mat))/2; %total weight
419
420  MOD = 0;
421  COMu = unique(C); %list of communities without repetiotions and in
         sorted order
422  %for each community I calculate modularity and then sum all together
423  for j=1:length(COMu)
424      Cj = (C==COMu(j)); %list of nodes in Cj
425      Ec = sum(sum(Mat(Cj,Cj))); %sum of weights between nodes in Cj
426      Et = sum(sum(Mat(Cj,:))); %sum of weights of nodes incident in nodes
             of Cj
427      if Et>0
```

```
428             MOD = MOD + Ec/(2*m)-(Et/(2*m))^2;
429         end
430     end
431
432     end
433
434     % Re-index community partition by size
435     function [C Ss] = reindex_com(COMold)
436     C = sparse(1,length(COMold));
437     COMu = unique(COMold);
438     S = sparse(1,length(COMu));
439     for l=1:length(COMu)
440         S(l) = length(COMold(COMold==COMu(l)));
441     end
442     [Ss INDs] = sort(S,'descend');
443     for l=1:length(COMu)
444         C(COMold==COMu(INDs(l))) = l;
445     end
446     end
447     %Re-index community partition not by size but by initial order
448     %function [C] = reindex_com(COMold)
449     %C = sparse(1,length(COMold)); %vector with a index for each node
450     %COMu = unique(COMold); %unique(v)=same data of v but not repetitions
        and in sorted order
451     %COMolds=sort(COMold);
452     %COMu=COMolds([true;diff(COMolds(:))>0]);
453     %for l=1:length(COMu)
454     %    C(COMold==COMu(l)) = l; %Riname the communities
455     %end
456     %end
```

Bibliography

[1] L. Euler, "Commentarii academiae scientiarum petropolitanae," *Solutio problematis ad geometriam situs pertinentis*, vol. 8, pp. 128–140, 1736.

[2] J. S. Coleman, *An Introduction to Mathematical Sociology*. Collier-Macmillan, London, UK, 1964.

[3] B. Krishnamurthy and J. Wang, "On network-aware clustering of web clients," in *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 97–110, 2000.

[4] R. Gallotti and M. Barthelemy, "The multilayer temporal network of public transport in great britain," *Scientific data*, vol. 2, no. 1, pp. 1–8, 2015.

[5] J. Chen and B. Yuan, "Detecting functional modules in the yeast protein–protein interaction network," *Bioinformatics*, vol. 22, no. 18, pp. 2283–2290, 2006.

[6] Y. Dourisboure, F. Geraci, and M. Pellegrini, "Extraction and classification of dense communities in the web," in *Proceedings of the 16th international conference on World Wide Web*, pp. 461–470, 2007.

[7] R. Guimerá and L. Amaral, "Cartography of complex networks: modules and universal roles j stat mech p02001," 2005.

[8] M. Steyvers *et al.*, "Kdd'04: Proceedings of the tenth acm sigkdd international conference on knowledge discovery and data mining," *New York, NY, USA: ACM*, pp. 306–315, 2004.

[9] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Stat mech theory exp. 2008," *P10008*, 2008.

[10] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[11] S. Fortunato and M. Barthelemy, "Resolution limit in community detection," *Proceedings of the national academy of sciences*, vol. 104, no. 1, pp. 36–41, 2007.

[12] J. Ruan and W. Zhang, "Identifying network communities with a high resolution," *Physical Review E*, vol. 77, no. 1, p. 016104, 2008.

[13] B. H. Good, Y.-A. De Montjoye, and A. Clauset, "Performance of modularity maximization in practical contexts," *Physical Review E*, vol. 81, no. 4, p. 046106, 2010.

[14] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[15] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," in *Parallel Numerical Algorithms*, pp. 323–368, Springer, 1997.

[16] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[17] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," in *Classic papers in combinatorics*, pp. 243–248, Springer, 2009.

[18] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *Journal of the ACM (JACM)*, vol. 35, no. 4, pp. 921–940, 1988.

[19] G. W. Flake, S. Lawrence, C. L. Giles, and F. M. Coetzee, "Self-organization and identification of web communities," *Computer*, vol. 35, no. 3, pp. 66–70, 2002.

[20] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1. Springer series in statistics New York, 2001.

[21] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, pp. 281–297, Oakland, CA, USA, 1967.

[22] W. E. Donath and A. J. Hoffman, "Lower bounds for the partitioning of graphs," in *Selected Papers Of Alan J Hoffman: With Commentary*, pp. 437–442, World Scientific, 2003.

[23] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak mathematical journal*, vol. 23, no. 2, pp. 298–305, 1973.

[24] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[25] P. Holme, M. Huss, and H. Jeong, "Subnetwork hierarchies of biochemical pathways," *Bioinformatics*, vol. 19, no. 4, pp. 532–538, 2003.

[26] J. Pinney and D. Westhead, "Betweenness-based decomposition methods for social and biological networks. interdisciplinary statistics and bioinformatics, 87–90," 2006.

[27] E. Estrada, "Community detection based on network communicability," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 21, no. 1, p. 016103, 2011.

[28] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[29] M. E. Newman, "Fast algorithm for detecting community structure in networks," *Physical review E*, vol. 69, no. 6, p. 066133, 2004.

[30] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[31] L. Danon, A. Díaz-Guilera, and A. Arenas, "The effect of size heterogeneity on community identification in complex networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2006, no. 11, p. P11010, 2006.

[32] P. Schuetz and A. Caflisch, "Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement," *Physical Review E*, vol. 77, no. 4, p. 046112, 2008.

[33] K. Kirkpatrick, C. Gelatt, and M. Vecchi, "Science 220 671 kirkpatrick ks 1984," *J. Stat. Phys*, vol. 34, p. 975, 1983.

[34] R. Guimerá, M. Sales-Pardo, and L. A. N. Amaral, "Modularity from fluctuations in random graphs and complex networks," *Physical Review E*, vol. 70, no. 2, p. 025101, 2004.

[35] S. Boettcher and A. G. Percus, "Optimization with extremal dynamics," *complexity*, vol. 8, no. 2, pp. 57–62, 2002.

[36] J. Duch and A. Arenas, "Community detection in complex networks using extremal optimization," *Physical review E*, vol. 72, no. 2, p. 027104, 2005.

[37] V. A. Traag, P. Van Dooren, and Y. Nesterov, "Narrow scope for resolution-limit-free community detection," *Physical Review E*, vol. 84, no. 1, p. 016114, 2011.

[38] L. Donetti and M. A. Muñoz, "Detecting network communities: a new systematic and efficient algorithm," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2004, no. 10, p. P10012, 2004.

[39] N. A. Alves, "Unveiling community structures in weighted networks," *Physical Review E*, vol. 76, no. 3, p. 036101, 2007.

[40] A. Capocci, V. D. Servedio, G. Caldarelli, and F. Colaiori, "Detecting communities in large networks," *Physica A: Statistical Mechanics and its Applications*, vol. 352, no. 2-4, pp. 669–676, 2005.

[41] B. Yang and J. Liu, "Discovering global network communities based on local centralities," *ACM Transactions on the Web (TWEB)*, vol. 2, no. 1, pp. 1–32, 2008.

[42] D. Fasino and F. Tudisco, "An algebraic analysis of the graph modularity," *SIAM Journal on Matrix Analysis and Applications*, vol. 35, no. 3, pp. 997–1018, 2014.

[43] D. Fasino and F. Tudisco, "Generalized modularity matrices," *Linear Algebra and its Applications*, vol. 502, pp. 327–345, 2016.

[44] F.-Y. Wu, "The potts model," *Reviews of modern physics*, vol. 54, no. 1, p. 235, 1982.

[45] B. D. Hughes, *Random walks and random environments: random walks*, vol. 1. Oxford University Press, 1995.

[46] A. Pikovsky, J. Kurths, M. Rosenblum, and J. Kurths, *Synchronization: a universal concept in nonlinear sciences*, vol. 12. Cambridge university press, 2003.

[47] D. J. MacKay and D. J. Mac Kay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[48] R. Winkler, *Introduction to Bayesian Inference and Decision*. Probabilistic Publishing, Gainesville, 2003.

[49] P. Doreian, V. Batagelj, and A. Ferligoj, *Generalized blockmodeling*, vol. 25. Cambridge university press, 2005.

[50] F. Lorrain and H. C. White, "Structural equivalence of individuals in social networks," *The Journal of mathematical sociology*, vol. 1, no. 1, pp. 49–80, 1971.

[51] M. G. Everett and S. P. Borgatti, "Regular equivalence: General theory," *Journal of mathematical sociology*, vol. 19, no. 1, pp. 29–52, 1994.

[52] D. R. White and K. P. Reitz, "Graph and semigroup homomorphisms on networks of relations," *Social Networks*, vol. 5, no. 2, pp. 193–234, 1983.

[53] K. Burnham *et al.*, "dr anderson. 2002. model selection and multimodel inference: a practical information-theoretic approach," *Ecological Modelling. Springer Science & Business Media, New York, New York, USA*.

[54] E. Ziv, M. Middendorf, and C. H. Wiggins, "Information-theoretic approach to network modularity," *Physical Review E*, vol. 71, no. 4, p. 046117, 2005.

[55] N. Tishby, F. Pereira, and W. Bialek, "Proceedings of the 37th annual allerton conference on communication, control and computing," 1999.

[56] P. Paatero and U. Tapper, "Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values," *Environmetrics*, vol. 5, no. 2, pp. 111–126, 1994.

[57] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788–791, 1999.

[58] P. O. Hoyer, "Non-negative matrix factorization with sparseness constraints," *Journal of machine learning research*, vol. 5, no. Nov, pp. 1457–1469, 2004.

[59] J. Kim and H. Park, "Toward faster nonnegative matrix factorization: A new algorithm and comparisons," in *2008 Eighth IEEE International Conference on Data Mining*, pp. 353–362, IEEE, 2008.

[60] I. Kotsia, S. Zafeiriou, and I. Pitas, "A novel discriminant non-negative matrix factorization algorithm with applications to facial image characterization problems," *IEEE Transactions on Information Forensics and Security*, vol. 2, no. 3, pp. 588–595, 2007.

[61] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[62] D. Cai, X. He, X. Wang, H. Bao, and J. Han, "Locality preserving nonnegative matrix factorization.," in *IJCAI*, vol. 9, pp. 1010–1015, 2009.

[63] Q. Gu and J. Zhou, "Neighborhood preserving nonnegative matrix factorization.," in *BMVC*, pp. 1–10, 2009.

[64] T. Bühler and M. Hein, "Spectral clustering based on the graph p-laplacian," in *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 81–88, 2009.

[65] X. Bresson, T. Laurent, D. Uminsky, and J. Von Brecht, "Multiclass total variation clustering," in *Advances in Neural Information Processing Systems*, pp. 1421–1429, 2013.

[66] H. Hu, T. Laurent, M. A. Porter, and A. L. Bertozzi, "A method based on total variation for network modularity optimization using the mbo scheme," *SIAM Journal on Applied Mathematics*, vol. 73, no. 6, pp. 2224–2246, 2013.

[67] Z. M. Boyd, E. Bae, X.-C. Tai, and A. L. Bertozzi, "Simplified energy landscape for modularity using total variation," *SIAM Journal on Applied Mathematics*, vol. 78, no. 5, pp. 2439–2464, 2018.

[68] F. Tudisco, P. Mercado, and M. Hein, "Community detection in networks via nonlinear modularity eigenvectors," *SIAM Journal on Applied Mathematics*, vol. 78, no. 5, pp. 2393–2419, 2018.

[69] F. Tudisco and D. J. Higham, "A nonlinear spectral method for core–periphery detection in networks," *SIAM Journal on Mathematics of Data Science*, vol. 1, no. 2, pp. 269–292, 2019.

[70] A. Cristofari, F. Rinaldi, and F. Tudisco, "Total variation based community detection using a nonlinear optimization approach," *SIAM Journal on Applied Mathematics*, vol. 80, no. 3, pp. 1392–1419, 2020.

[71] M. A. Rodriguez and J. Shinavier, "Exposing multi-relational networks to single-relational network analysis algorithms," *Journal of Informetrics*, vol. 4, no. 1, pp. 29–41, 2010.

[72] A. Lancichinetti and S. Fortunato, "Consensus clustering in complex networks," *Scientific reports*, vol. 2, p. 336, 2012.

[73] J. Kim and J.-G. Lee, "Community detection in multi-layer graphs: A survey," *ACM SIGMOD Record*, vol. 44, no. 3, pp. 37–48, 2015.

[74] H. Li, Z. Nie, W.-C. Lee, L. Giles, and J.-R. Wen, "Scalable community discovery on textual data with relations," in *Proceedings of the 17th ACM conference on Information and knowledge management*, pp. 1203–1212, 2008.

[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*, pp. 534–545, IEEE, 2012.

[76] Y. Zhou, H. Cheng, and J. X. Yu, "Graph clustering based on structural/attribute similarities," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 718–729, 2009.

[77] Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng, "A model-based approach to attributed graph clustering," in *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pp. 505–516, 2012.

[78] A. Silva, W. Meira Jr, and M. J. Zaki, "Mining attribute-structure correlated patterns in large attributed graphs," *arXiv preprint arXiv:1201.6568*, 2012.

[79] Y. Ruan, D. Fuhry, and S. Parthasarathy, "Efficient community detection in large networks using content and links," in *Proceedings of the 22nd international conference on World Wide Web*, pp. 1089–1098, 2013.

[80] W. Tang, Z. Lu, and I. S. Dhillon, "Clustering with multiple graphs," in *2009 Ninth IEEE International Conference on Data Mining*, pp. 1016–1021, IEEE, 2009.

[81] X. Dong, P. Frossard, P. Vandergheynst, and N. Nefedov, "Clustering with multi-layer graphs: A spectral perspective," *IEEE Transactions on Signal Processing*, vol. 60, no. 11, pp. 5820–5831, 2012.

[82] Z. Zeng, J. Wang, L. Zhou, and G. Karypis, "Coherent closed quasi-clique discovery from large dense graph databases," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 797–802, 2006.

[83] J. Pei, D. Jiang, and A. Zhang, "On mining cross-graph quasi-cliques," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pp. 228–238, 2005.

[84] S. Paul, Y. Chen, *et al.*, "Spectral and matrix factorization methods for consistent community detection in multi-layer networks," *The Annals of Statistics*, vol. 48, no. 1, pp. 230–250, 2020.

[85] D. Zhou and C. J. Burges, "Spectral clustering and transductive learning with multiple views," in *Proceedings of the 24th international conference on Machine learning*, pp. 1159–1166, 2007.

[86] P.-Y. Chen and A. O. Hero, "Multilayer spectral graph clustering via convex layer aggregation: Theory and algorithms," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 3, no. 3, pp. 553–567, 2017.

[87] T. P. Peixoto, "Bayesian stochastic blockmodeling," *Advances in network clustering and blockmodeling*, pp. 289–332, 2019.

[88] J. D. Wilson, J. Palowitch, S. Bhamidi, and A. B. Nobel, "Community extraction in multilayer networks with heterogeneous community structure," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 5458–5506, 2017.

[89] A. Kumar and H. Daumé, "A co-training approach for multi-view spectral clustering," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 393–400, 2011.

[90] A. Kumar, P. Rai, and H. Daume, "Co-regularized multi-view spectral clustering," in *Advances in neural information processing systems*, pp. 1413–1421, 2011.

[91] V. Pareto, "Cours d'économie politique, rouge," *Lausanne, Switzerland*, 1896.

[92] A. Scherrer, *Matlab / C++ implementation of community detection algorithm*, 28 September 2010. `https://github.com/jblocher/matlab-network-utilities/tree/master/Louvain`.

[93] L. Danon, A. Diaz-Guilera, J. Duch, and A. Arenas, "Comparing community structure identification," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2005, no. 09, p. P09008, 2005.

[94] J. Liu, C. Wang, J. Gao, and J. Han, "Multi-view clustering via joint nonnegative matrix factorization," in *Proceedings of the 2013 SIAM International Conference on Data Mining*, pp. 252–260, SIAM, 2013.

[95] D. Greene and P. Cunningham, "A matrix factorization approach for integrating multiple data views," in *Joint European conference on machine learning and knowledge discovery in databases*, pp. 423–438, Springer, 2009.

[96] N. Rasiwasia, J. Costa Pereira, E. Coviello, G. Doyle, G. R. Lanckriet, R. Levy, and N. Vasconcelos, "A new approach to cross-modal multimedia retrieval," in *Proceedings of the 18th ACM international conference on Multimedia*, pp. 251–260, 2010.

[97] P. Mercado, F. Tudisco, and M. Hein, "Generalized matrix means for semi-supervised learning with multilayer graphs," in *Advances in Neural Information Processing Systems*, pp. 14877–14886, 2019.