University of Bergen
Department of informatics

A new approach for finding communities of edges in complex networks

Author: Morten Movik

Supervisor: Christophe Crespelle

UNIVERSITETET I BERGEN

Det matematisk-naturvitenskapelige fakultet

November, 2019

Abstract

Discovering dense subparts, called communities, in complex networks is a fundamental issue in data analysis. A popular way to do this is to create a partition of the network. This partition can either be a partition of nodes, or a partition of edges. In this thesis I propose a new approach to finding a partition of the edges, by mimicking the approach of the Louvain algorithm, one of the most popular methods for node partitions. The Louvain algorithm is a greedy optimization technique using modularity as an objective function. I propose several different objective functions, edge modularities, to optimize in this approach and test the algorithm with different edge modularities on real networks.

Contents

Chapter 1

Introduction

1.1 Community Detection

Community detection is about finding dense subparts in graphs called communities. Unfortunately there is not one formal and general definition of what a good community is. Intuitively we want many edges to be between nodes belonging to the same community, and few edges whose endpoints does not belong to the same community. As an example, consider a social network, a graph where each node represents a person and each edge is a tie between two people. Examples of communities in this graph can be a family, a group of friends, and a football team. Graphs representing real system often have a community structure, meaning it's possible to find good communities in the graph. Discovering this community structure is an important field of study, and a lot of research has been done on community detection [10].

Community detection algorithms can be exact, finding the "best" communities according to some measure. However, independently of the chosen formal definition of a good community, this often turns out to be a NP-complete problem, and we are often interested in finding community structure in large networks. For instance we can find communities by using cluster editing, where the goal is to find the minimum number of edits that makes the graph a disjoint union of cliques [2]. One edit is removing an edge or adding an edge. This problem turns out to be NP-complete [6]. Because we are often interested in finding communities in large networks, we need an algorithm that is efficient. The problem can be solved with fixed parameter tractable algorithms, which have running time $f(k) * n^c$ for some constant c and some parameter k, however $f(k)$ is typically some exponential function. The problem could potentially be solved with an approximation algorithm, giving a solution that's guaranteed to be within some constant factor of the optimal. However, as far as I know, there does not exist any efficient approximation algorithm for this problem. This is why many community detection algorithms are heuristics, algorithms that can find good solutions, for example by optimizing some objective function, but have no guarantee for how good the solutions are. In this thesis I will focus on heuristics.

Many community detection approaches focus on creating a partition $\mathscr{C} = \{C_0, C_1, C_2, ..., C_N\}$ of the nodes in a graph, meaning that $C_i \cap C_j = \emptyset$ for any i,j. In other words each node belongs to one and only one community. Communities can also be overlapping however. In overlapping communities we also divide the graph into communities $C_0, C_1, C_2, ..., C_N$, but the communities can overlap with each other, meaning it's possible that $C_i \cap C_j \neq \emptyset$ for some i, j. A third option is to create a partition of the edges in the graph. I will do an algorithm for finding an edge partition in a network, to do this I will mimic the approach of the Louvain algorithm, which makes a partition of the nodes i a graph.

1.2 Some Approaches For Communities of Nodes

In this section I will mention some popular approaches for community detection finding communities of nodes. Most of them are for partitions of nodes.

Hierarchical Clustering, Hastie et. al. [11]. Sometimes a graph can contain a hierarchy of communities. As an example let's consider a social network of all students in a city. In this graph each school can be one community. Students that go to the same school are more likely to know each other than students going to different schools. But within each school we can also have one community for each class as well. To find communities like these, where small communities are included in larger communities, we can use hierarchical clustering. To decide which nodes belong in the same community, hierarchical clustering uses a similarity measure. Every pair of nodes in the graph receives a value of this measure, indicating how similar they are. And the algorithm aims to create communities where nodes inside the same community have a high similarity to each other. There are two categories of hierarchical clustering algorithms, based on how they group nodes with high similarity. *Agglomerative algorithms*, which iteratively merge clusters if their similarity is high enough. And *Divisive algorithms*, that iteratively removes edges connecting nodes with low similarity.

Partitional clustering (e.g. [16]) Partitional clustering techniques finds a preassigned number of clusters, k, in a set of data points. The data points, or nodes, are embedded in a metric space such that they have some distance measure between them. This distance measure is a measure of dissimilarity between nodes. Then the nodes are separated into k clusters, with the goal of minimizing or maximizing some cost function based on the distance between nodes or centroids. One of the most popular techniques using partitional clustering is *k-means clustering* by MacQueen [16], which uses the squared error function as a cost function.

Spectral clustering [8]. Given a number of objects (for instance nodes), let S be a symmetric, non-negative similarity function defined for every pair of objects. Spectral clustering are techniques creates a partition of the set into clusters by using the eigenvector of S or matrices derived from S. This involves translating the original objects into a set of points in space, where the coordinates of these are elements of eigenvectors. These coordinates are then clustered using techniques like *k-means clustering*.

Newman and Girvan [18] Newman and Girvan introduced an approach similar to divisive hierarchical clustering techniques. However instead of using a similarity measure describing whether two nodes should be in the same community, Newman and Girvan uses a *betweenness* measure, describing weather an edge should connect two different communities or not. Then they remove edges one by one, dividing the network into smaller components. The process can be stopped at any stage, taking the components at that stage to be the communities. They then introduce modularity as a measure of the quality of a partition, and use this to see where the algorithm should stop.

Modularity Optimization [18], [5]. Modularity was introduced in 2004 by Newman and Girvan [18]. It is a function that can tell us something about how good a partition is. Modularity has become a popular tool in the field of community detection. It works by counting the number of edges with both endpoints inside the same community, and then comparing this to the expected number of edges with both endpoints inside the same community in a random graph (section 2.1.1). One of the most popular approaches to community detection is modularity optimization. Finding the maximum possible modularity in a graph is NP-complete [7], but there are many methods that does a good job of finding high values of modularity in a more reasonable amount of time. One such method is the Louvain algorithm, which appears to run in linear time on most real datasets [5]. The Louvain algorithm [5] is probably the most successfull heuristic for finding a partition of nodes. The algorithm is a greedy optimization method using modularity. In the algorithm each node starts off in it's own community. It works by iteratively moving nodes to a community that gives the highest increase in modularity. The Louvain algorithm will be discussed more in section 2.1

Clique Percolation, Palla et. al. [19]. There are also popular algorithms for finding overlapping communities. One of the most popular methods for overlapping communities is clique percolation. It is based on the idea that nodes inside the same community are likely to form a clique with each other, because of the high density of edges inside communities. Nodes that are in different communities are less likely to have edges between them, and are less likely to form a clique with each other. They use the term k-clique to indicate a clique with k nodes. Two cliques are considered *adjacent* if they share $k-1$ vertices. The algorithm starts out with some k-cliques as communities. It then grows the communities by merging adjacent k-cliques. Because one node can be involved in several k-cliques, this method produces overlapping communities.

1.3 Some Approaches for Communities of Edges

Consider a social network, it makes sense for one person to be a part of a family, a football team, and a workplace. If we want to create a node partition of this graph, then this person can only belong to one community, when it would make more sense for him/her to be part of all three communities. In this example it might make more sense to create a partition of the edges. That way we still have one community for the family, one for the football team, and one for the workplace, but one person can be related to all three. Consider $V(C)$ to be the nodes in G with an edge from community C incident to it. I will refer to $V(C_1) \cap V(C_2) \cap ...V(C_k)$ as the border between the communities C_1, C_2, ..., C_k. The person in the example above, is on the border between the three communities.

UELC, He et. al. [12]. Dongxiao He et. al. developed an algorithm that splits the graph into a partition with two edge communities. To do this it uses a link-node-link random walk, as well as markov dynamics. The algorithm then decides whether or not to accept each community based on a method using link density. Then on each of the two subgraphs induced by the new communities, it recursively repeats this process, dividing each subgraph into two edge-communities and deciding whether or not to accept them.

Evans et. al.[9] Evans et. al. introduced a method for finding link-partitions using the line graph and the Louvain algorithm. They first find the line graph corresponding to the original graph. Then they assign weights on the edges by using the concept of a random walker. The weights say something about how densely connected different nodes in the line graph are. Then they apply the Louvain algorithm to the line graph. The result is a node partition of the line graph, which corresponds to an edge partition in the original graph. This algorithm will be discussed further in section 2.2.

Ahn et. al.[3] The algorithm developed by Ahn et. al. use hierarchical clustering with a similarity measure for pairs of edges to build a dendrogram. Each leaf in the dendrogram represent an edge from the original graph. each branch of the dendrogram represent a community. Partitions of the graph into edge-communities can be found by cutting the dendrogram at various levels. Each branch in the cut is one community in the partition. To choose where to cut the dendrogram Ahn et. al. uses an objective function based on link-density.

Li et. al. [15] Li et. al. Formulates an objective function based on partition-density of edge communities and develops an integer linear programming model of the community detection problem. They then use a genetic algorithm to solve the integer programming model.

LMBP, He et. al.[14] He et. al. formulates a stochastic model called the link-model, LM. This model takes into account the varying sizes of the communities when describing community structure. They then use a maximum likelihood method to learn the parameters of LM. Then they use a scheme of iterative bipartition.

He et. al. [13] He et. al. introduces a mixture of node and link communities called hybrid node-link communities. In this scheme communities can be either node communities or link communities. In a graph with hybrid node-link communities, a node can belong to a node-community and/or it can have an edge from an edge-community incident to it.

1.4 The Goal of this Thesis: Link Partition in Static Networks Based on Edge Modularity

Modularity has become a very popular tool for node partitions. And one of the most successful algorithms for finding node partitions, the Louvain algorithm, is a method optimizing modularity. The idea behind this thesis is to provide a community detection algorithm for edge partitions by mimicking the approach of the Louvain algorithm. In order to do this it is necessary to formulate a modularity that works for edge-partitions.

The algorithm by Evans et. al. [9] (section 2.2) also uses the Louvain algorithm in their approach to finding an edge partition. However they do this by applying weights to the line graph and then running the Louvain algorithm directly on the line graph. These weights are based on local information, and say something about which edges from the original graph should be in a community together. This means that which edges end up in the same community, is not only decided by the optimization of modularity. It depends on the weights that were applied to the line graph. In this thesis I attempt to provide a global edge modularity, and mimic the Louvain approach in order to optimize this measure directly. I would also like to do this in a way that can be adapted to dynamic networks without too much difficulty.

In chapter 2 I describe my implementation the Louvain algorithm [5] as well as the algorithm by Evans et. al. [9] which can serve as a comparison to the results of my algorithm.

In chapter 3 I provide some definitions for an edge modularity. In chapter 4 I present the results of my algorithm with each of three different edge modularities, as well as the results of the first two methods, C and D, developed by Evans et. al. in [9], and the Louvain algorithm [5] on the same data.

Throughout this paper, unless otherwise specified, I will assume that graphs are undirected and unweighted. To refer to a pair of nodes, where the order of the nodes does not matter, I will use the shorthand uv, in other words $uv = \{u, v\}$. This means I will sometimes write $uv \in E$ to denote an undirected edge in a graph $G = \{V, E\}$ To denote the number of nodes $|V|$ in the graph, I will use n, to denote the number of edges $|E|$ I will use m.

Chapter 2

Implementing Existing Methods

In order to familiarize myself with existing methods, I have implemented them myself. In particular, I implemented the Louvain-algorithm [5], and the methods from [9]. The code can be found in appendix A, where the Louvain algorithm is in the same program as the algorithm for Evans et. al. [9] (section 2.2). The part of the algorithm that is the Louvain algorithm is about 700 lines, while the additional part required for the algorithm by Evans et. al. is about 350 lines. Some results of my implementation of these two algorithms can be found in table 4.5.

2.1 The Louvain Algorithm

The Louvain algorithm is a heuristic that works by optimizing the modularity function:

$$Q = \frac{1}{2m} \sum_{i,j \in V} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{2.1}$$

where m is the number of edges in the graph, A_{ij} is the weight of the edge between i and j, k_i is the total weight of edges connected to i, c_i is the community to which the node i belongs, δ is the Kronecker delta:

$$\delta(c_i, c_j) = \begin{cases} 0 & \text{if } c_i \neq c_j \\ 1 & \text{if } c_i = c_j \end{cases} \tag{2.2}$$

One strength of the Louvain algorithm is that the change in this modularity can be calculated in constant time. The change in modularity from moving an isolated node i into a community C can be calculated with:

$$\Delta Q = \left[\frac{\Sigma_{in} + 2k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m} \right)^2 \right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m} \right)^2 - \left(\frac{k_i}{2m} \right)^2 \right] \tag{2.3}$$

where Σ_{in} is the sum of the weights of links between nodes inside C, and Σ_{tot} is the sum of the weights of all links connected to some node in C.

The Louvain algorithm works by initially placing every node into its own community. It then loops through each node, checks the gain in modularity from placing it into the community of a neighbour instead of it's own community. The node is then placed in the community that provides the highest gain in modularity if that gain is positive, if the gain is negative it stays in the same community.

It keeps looping through nodes like this until it has gone for an entire loop over all the nodes without moving any node to a different community (all modularity gains were negative). At this point one stage of the algorithm is done. For the next stage it transforms the graph by contracting each community into one node. Nodes in this new graph have an edge between them if nodes inside the communities they were made from had edges between them. The number of edges that was between the communities are now weights on the edges between the nodes.

2.1.1 The Random Experiment in the Louvain Algorithm.

The term $-\frac{k_i k_j}{2m}$ in the modularity function is actually a comparison to a random experiment. The modularity is a comparison between how many edges are inside communities $\frac{1}{2m} \sum_{i,j \in V} A_{ij} \; \delta(c_i, c_j)$, and how many would be inside if the graph was constructed in a random way $\frac{1}{2m} \sum_{i,j \in V} \frac{k_i k_j}{2m} \; \delta(c_i, c_j)$. This random graph is constructed by using the configuration model [17], it fixes the communities, as well as the degrees of each node. We can visualize the graph as a collection of nodes, and connected to each node i are k_i edge-stubs that are not connected to anything yet. Then we choose two edge-stubs at random and connect them. Observe that there is a chance we will connect a node to itself, creating a self-loop, or connect the same two nodes multiple times, creating multiedges. However in typical small-density networks this will happen so rarely that it will not significantly alter the result.

2.1.2 My Implementation of the Louvain Algorithm [5]

I implemented the Louvain Algorithm from scratch, the code is included in Appendix A. The implementation achieves the same partition as [5] on the karate-club data, except one node is in a different community. This might be because of the order in which the nodes are considered. I considered the nodes in random order, and ran the program a few times to get this result. The modularity from my implementation when run on two larger datasets, were different from the ones obtained by [5], see table 2.1. Arxiv in the table below is a network of papers on arxiv [1] and web nd.edu [4] is a network of a subdomain of the internet. More results can be seen in table 4.5.

Dataset	#Nodes/#edges	from [5]	my implementation
Karate	34/78	0.42	0.42
Arxiv	9k/24k	0.813	0.935
web nd.edu	325k/1M	0.622	0.963

Table 2.1: The modularity obtained with my implementation of the Louvain algorithm, and the modularities presented in [5]

2.1.3 Criticism

The modularity of Newman and Girvan [18] is very popular. However it might be worth mentioning some possible downsides to this quality function. It tends to generate large communities, and miss smaller ones. And if it's given a graph that consists of nothing but one clique, it will still prefer a partition with more than one community. If the algorithm is applied to a large grid, it will also partition it into several communities, even though there is no naturally denser parts. Despite all of this, it's still one of the most successful ways to judge the quality of a partition.

2.2 T. S. Evans et al.

Evans et. al. [9] uses the Louvain algorithm as it is, but changes the input graph G. It does this in several different ways.

Using the Line Graph, C. The first method used in [9] is based on the line graph. They call the adjacency matrix of the line graph C. In this new graph $G(C)$ each edge of the original graph is represented by a node. If two edges in the original graph shared a node, they have an edge between them in $G(C)$. Let B be the incidence matrix, $B_{i\alpha} = 1$ if node i has edge *alpha* incident to it, otherwise $B_{i\alpha=0}$.

$$C_{\alpha\beta} = \sum_{i} B_{i\alpha} B_{i\beta} (1 - \delta_{\alpha\beta})$$

Line Graph with Weights, D. The next graph used in [9], $G(D)$, is the same as $G(C)$ but with weights. Evans et. al. uses a link-node-link to derive the weights. Two edges $\alpha = uv$ and $\beta = vw$ in the original graph are connected by an edge in the line graph (because they share the node v). A random walker located on the edge uv can move to any other neighbour of either u or v with equal probability. If the random walker moves through the node i, the probability it chooses to walk to vw is $\frac{1}{k_i - 1}$. Because of this the edge $\alpha\beta$ in the line graph will have weight $\frac{1}{k_i - 1}$.

$$D_{\alpha\beta} = \sum_{i} \frac{B_{i\alpha} B_{i\beta}}{k_i - 1} (1 - \delta_{\alpha\beta})$$

Line Graph with Weights Based on a Projection of a Node Random Walk, E_1 The previous method is based on the idea of a random walk on the line graph. This can't be related to a random walk on nodes, because link-node-link walker can move through the same node v on two subsequent steps. If we try to interpret this random walk on edges as a random walk on nodes, it will look like a self loop. E1 is based on the idea of a random walk on nodes that is projected onto edges. They first assume that all neighbouring links of some node i are connected in the line graph with weight $\frac{1}{k_i}$. This is leads to an adjacency matrix:

$$E_{\alpha\beta} = \sum_{i, k_i > 0} \frac{B_{i\alpha} B_{i\beta}}{k_i}$$

This is considered to be the state when the random walker is located on a node, but nod moved yet. The adjacency matrix E_1 obtained after the walker moves, can be calculated using: $E_1 = EE - E$.

Results of my implementation of the algorithm can be seen in table 4.5. Unfortunately I have not been able to implement method E_1 because of a segmentation fault.

Chapter 3

A New Approach for Link Partitions

3.1 What is a Good Partition

A good partition of nodes. The Louvain modularity counts how many edges are inside a community. Then it compares to how many edges would be inside in a random graph. Instead of counting how many edges are inside a community however, we could also count how few are between communities. This means that intuitively a good node-partition is one where the subgraphs induced by the communities looks like cliques, and there are few edges between communities.

In order to come up with a good measure for a partition, it can be useful to think about what a graph with a perfect node-partition would look like. A good measure will then tell us something about how far we are from this perfect situation.

A perfect situation for a partition of nodes would be a union of disjoint cliques. See figure 3.1 for an example of a perfect partition nodes. The colors represent communities, there is a red, a green, and a blue community.

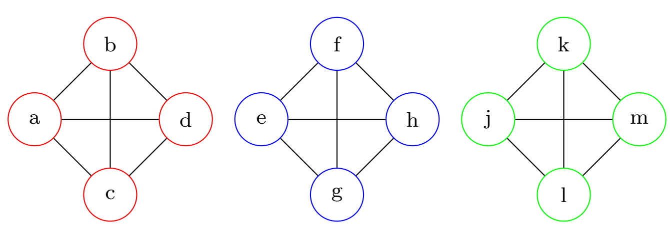

Figure 3.1: An example of a perfect partition on an ideal graph.

A good partition of edges. In an edge-partition, every edge is inside one community, so it doesn't make sense to count how many edges are inside communities. However we can still require that the subgraphs induced by the communities look like cliques. And we will see that a consequence of the communities looking like cliques, is that the number of nodes on the border between communities must be small. To illustrate, let's look at the perfect partition for an edge-partition. For the perfect edge-partition we can try something similar to what we did with nodes, and define a graph with a perfect partition as a graph where the subgraph induced by $V(C_i)$ is a clique for all i, where $V(C_i)$ is the set of nodes that are connected to some edge in C_i. A consequence of this is that $V(C_i) \cap V(C_j) \leq 1$, in other words, only one node can lie on the border between two specific communities. See figure 3.2 for an example.

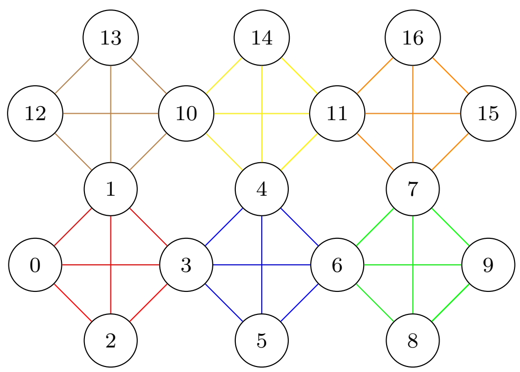

Figure 3.2: An example of a perfect link-partition on an ideal graph.

3.2 Overview of Measures

To achieve a good edge-partition, we want the communities to look like cliques. In the perfect partition the communities are all cliques, so a natural approach to create a measure is to try to create one that says something about how far away we are from a clique. But there is also another way to think about the problem. Notice that in order for an edge-partition to be perfect, there can only be one node on the border between two specific communities. If two communities share more than one node, then the communities are not cliques, since an edge between two of these nodes can at most belong to one of the communities. This leads to the idea of minimizing the size of the border. This is the first approach that I have tried to follow. Unfortunately if the borders between communities are small, it doesn't mean that the communities look like cliques. So the measures I have tested are aiming to say something about how far away the communities are from cliques.

For each of these two criteria, there are several different ways to formalize a measure. To minimize the size of the border, I propose three different measures. Each measure minimizes something different.

border-based approaches

- border nodes
- border pairs
- border pairs without an edge

I propose two different measures that focus on making the communities look like cliques. item For all i, the subgraph induced by $V(C_i)$ looks like a clique. The following measures should be minimized:

clique-based approaches

- non-edges inside each community.
- number of pairs in each community.

3.3 Border Based Measures

3.3.1 Minimize Border Nodes

In the perfect situation for edge-partitions described above, each pair of communities only had at most one node between them, in other words $V(C_i) \cap V(C_j) \leq 1$ for each pair i, j. The number of nodes on the border is one possible measure we can minimize. Note that we may count one node several times if it is on the border between more than two communities.

This is because one node can be a problem for many pairs of communities, and it should then account for more than a node that's only between one pair of communities.

$$R_{nodes} = \frac{2}{|V||\mathscr{C}|(|\mathscr{C}|-1)} * \sum_{C_i, C_j \in \mathscr{C}} |(V(C_i) \cap V(C_j))|$$

There can't be more than $\frac{|V||\mathscr{C}|(|\mathscr{C}|-1)}{2}$ border-nodes since each node can at maximum be on the border between every community. So this measure will be between 0 and 1.

This measure doesn't feel quite right, since having such border-nodes is not necessarily a bad thing. Imagine a social network where the edges represent types of relationships between people. We might want one community bordering this node to be that person's colleagues, another might be his friends and yet another his family. It seems like what we really want might be to minimize the number of pairs on the border.

3.3.2 Minimize Border Pairs

If $V(C_i)$ and $V(C_j)$ both contain the same pair of nodes, then the partition is not perfect. If there is an edge between the pair, it can only belong to one community. So we are at least one edge away from the perfect situation. This measure will count the number of pairs that are shared between each pair of communities:

$$R_{pairs} = \frac{4}{|V|(|V|-1)(|\mathscr{C}| * (|\mathscr{C}|-1))} \sum_{C_i C_j \in \mathscr{C}} \frac{|\{V(C_i) \cap V(C_j)\}| \ * \ (|\{V(C_i) \cap V(C_j)\}| - 1)}{2} \tag{3.1}$$

Again note that a pair that lies on the border between more than two communities will be counted several times. Here $\frac{|V|(|V|-1)(|\mathscr{C}|*(|\mathscr{C}|-1))}{4}$ is to make sure the expression is between 0 and 1, it is a upper limit to how many pairs can be shared between communities. Every pair can at most belong to every community.

3.3.3 Minimize Border Pairs without an edge

If we simply count the number of pairs on the border, like in the previous measure, there are two possibilities for each pair: The pair has an edge between them, or it does not have an edge between them. As an example of a pair on the border between several communities, let's consider two people that are colleagues, play on the same football team and play in the same chess club. It would be strange if these two people did not know eachother. In other words, we would expect these two nodes to have an edge between them. If they do know eachother it's not strange for them to both be in some of the same communities. So perhaps instead of measuring simply the number of pairs on the border, it's better to restrict it to the number of pairs without an edge between them. This measure will minimize the number of pairs on the border that does not have an edge between them.

$$R_{border-non-edges} = \frac{4}{|V|(|V|-1)(|\mathscr{C}| * (|\mathscr{C}|-1))} \sum_{C_i C_j \in \mathscr{C}} |\{uv \mid u, v \in V(C_i) \cap V(C_j),\, uv \notin E\}|.$$

$\frac{4}{|V|(|V|-1)(|\mathscr{C}|*(|\mathscr{C}|-1))}$ is to make sure the expression is normalized. At most every pair is on the border between every community.

3.4 Clique based Measures

3.4.1 Minimize Non-edges Inside Communities

Consider the perfect partition, the subgraph induced by some $V(C_i)$ is a clique. In order to judge how far we are from the perfect situation, we can count how many non-edges are in the subgraph induced by each $V(C_i)$. This is similar to the modularity used in Louvain, which counts the number of edges inside communities. But instead of maximizing the number of edges inside communities this measure minimizes the number of missing edges from the subgraph induced by $V(C)$.

$$R_{non-edges} = \frac{1}{|\mathscr{C}| * |\bar{E}|} \sum_{C \in \mathscr{C}} |\{uv \mid u, v \in V(C), uv \notin E\}|$$

$|\mathscr{C}| * |\bar{E}|$ is a normalization factor, such that $0 \leq R_{non-edges} \leq 1$. This counts the number of non edges for each community. This means that if there is a non-edge between a pair of nodes uv, and uv are together on the border between several communities, then that non-edge will be counted several times. More precisely, a non-edge will be counted $|\{C_i \mid uv \in V(C_i)\}|$ times. Consider figure 3.3, for this partition, the number of non-edges are counted as 4, not 3, because nodes 2 and 3 are counted once for the red community **and** once for the blue community. One problem with this measure is that it would not care if one clique was separated into several communities, since every pair of nodes in each community still has an edge between them. See figure 3.4 for an example. This clique is divided into two communities, but still get a perfect score.

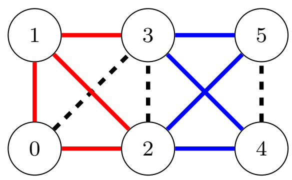

Figure 3.3: Edge-partition into a blue and red community. Dashed lines represent non-edges.

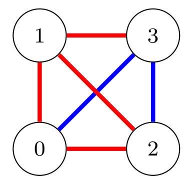

Figure 3.4: Edge-partition into a red and blue community

3.4.2 Minimize Number of Pairs in Each Community

Another idea to measure how far away the communities are from cliques is to is to count the number of pairs inside each community.

$$pairs(\mathscr{C}) = \sum_{C \in \mathscr{C}} \frac{|V(C)| * (|V(C)| - 1)}{2}. \tag{3.2}$$

The idea is that we want to put edges in communities where they do not contribute much to the number of pairs in that community. Let's say we want to know how much the number of pairs increases if we put an edge ab into a community C_0. The edge will not contribute to the number of pairs at all in C_0 if both $a, b \in V(C_0)$. If $a \in V(C_0)$ and $b \notin V(C_0)$ then the number of pairs increases by $|V(C_0)|$ (a makes one new pair with each other node in $V(C_0)$). If neither $a, b \notin V(C_0)$ then the number of pairs increases by $2|V(C_0| + 1$ (both a and b makes a new pair with every other node in $V(C_0)$ and ab itself is a new pair. Notice that if a pair ab will be counted several times if it's on the border between several communities, the same way a non-edge will be counted several times in section 3.4.1.

3.5 Random Experiment

The modularity used in Louvain counts the number of edges with both endpoints inside the same community (see section 2.1). If it didn't compare this to a random experiment, it would be trivial to obtain a node-partition that is perfect according to that measure. Just put everything inside one community. The measures proposed in this thesis have the same problem. Each one has a trivial perfect case, unless we compare to a random experiment.

For each of the border-based measures, a trivial partition that minimizes the measures is one where every edge is in the same community. That way the graph has only one community and there is no border. Since each of the border-based measures wants to minimize something on the border this is a perfect case according to each of those measures. For the measure in 3.4.1 the trivial case is to put every edge in it's own community. That way there are no non-edges inside any community.

The measure in 3.4.2, counts the number of pairs inside each community. Since every edge uv in the graph is contained in a community, it accounts for at least one pair (u and v). So a trivial way to minimize this measure is to put every edge in it's own community. That way the number of pairs inside communities are the same as the number of edges in the graph.

To avoid such trivial partitions we compare to the expected value of each measure in some random experiment. I will propose several possible experiments for comparison with a measure for edges.

3.5.1 Assign C_i Edges in a Random Graph to C_i For All $i < |\mathscr{C}|$

The first random experiment is one where we keep little information. Let's say we have an edge-partition $\mathscr{C} = \{C_0, C_1, C_2, ..., C_N\}$. We create a random graph like the one used in Louvain, except the first $|C_0|$ edges created by connecting edge-stubs belong to community C_0. The next $|C_1|$ edges belong to C_1 and so on.

We end up with a random graph like the one used in Louvain, and an edge-partition with communities where each community has the same size as in the original partition. But the edges are spread out randomly in a random graph. Because so little information about the original partition is kept, the experiment will not be as strongly related to the partition under investigation as we might like.

3.5.2 Keep the Degree of Each Edge's Endpoint

This experiment is a variation of 3.5.1 with one additional constraint. We keep the degrees of nodes incident to edges. In other words, if node v has degree 3 and node u has degree 2, then the edge uv can only be reassigned to a pair of nodes where one has degree 3 and the other has degree 2. Thus we keep more information and our experiment is more strongly related to the partition we compare to. However it might not be random enough for all inputs. If there is only one edge between nodes of degree 12 and degree 14, then that edge is guaranteed to still be there in the random experiment.

3.5.3 Keep Community-distribution of Endpoints

Another way to do the experiment that looks more like the one used in Louvain is to fix $V(C_i)$ and the degree of each node in the subgraph induced by $V(C_i)$. Then for each community C of size k we randomly assign k edges. Node u might have 3 edges in the red community and 2 edges in the blue community. We reshuffle the edges, but make sure u still has 3 edges in the red and 2 edges in the blue community, $k_{i,red} = 3$ and $k_{i,blue} = 2$.

It's easy to see the parallel to the experiment in the Louvain modularity. In the Louvain algorithm, the partition and degrees of every node is kept, and only the edges are moved. Here we keep all the $V(C_i)$ and then rearrange edges.

A problem with this experiment might be that we keep too much information. There might be too few ways to rearrange the communities in this way for it to be meaningful as a comparison.

3.5.4 Assign Communities to Edges Uniformly at Random

In this experiment we keep the graph as it is and instead randomly reassign edges to different communities. Given a graph and an edge partition, go through all the edges and assign a community to them. Choose each community C with probability $\frac{|C|}{|E|}$, where m is the total number of edges in the graph. An advantage of this method is that it can be fairly easy to work with. The problem is that the communities can end up being different in size from the communities we started with, so it's not as related to our initial partition as we would like.

The goal of this thesis is a new approach for community detection in complex networks, but a secondary goal, or a hope, is that this approach should be easy to adapt to dynamic networks. If the experiment we use changes the graph, it can be difficult to adapt to a dynamic network, since it is not clear how to address the time-aspect of the dynamic network. This experiment however can be done on a dynamic network the same way it's done on a static one.

3.6 Further Exploring *pairs*

The goal of this thesis was to mimic the Louvain approach, but for edges. The modularity used in Louvain does not look at the border between communities. It measures how far away the communities are from cliques by counting the number of edges inside the communities. Focusing on how similar a partition is to a clique also has the advantage that if a partition is similar to a partition of cliques, then the border is also small (as mentioned in 3.2). Because of this it makes sense to choose a measure that is also clique-based. Out of the two clique-based measures proposed, the one in 3.4.1 has the problem that if a clique is partitioned into two communities it will give a perfect score. So I have chosen the measure in 3.4.2. The random experiment I chose for this measure is the one in 3.5.4.

Normalizing

An intuitive way to compare this to the random experiment would be:

$$pairs - \mathbb{E}\left(pairs(\mathscr{C})\right)$$

But if we want to compare the results of this expression between different partitions with different graphs, it needs to be normalized. This is not so simple however since the experiment in 3.5.4 can end up creating communities of different sizes than the partition we compare to. So the expected value of $pairs$ using this experiment can have a different range of possible values than $pairs$. So how do we normalize this? Instead of normalizing, I will present two possible definitions of an edge modularity that circumvents this issue. The first is naturally normalized in the way it compares to random. The second does not normalize at all, this means values of the edge modularity is not meaningful to compare between graphs, but it might still provide good communities when employed in the algorithm.

Edge Modularity Inspired by Global Density

One way to formalize a measure using the number of pairs is to consider the concept of density. The density of a graph is the ratio of the number of edges in the graph to the number of pairs of nodes

$$\frac{2|E|}{|V| * (|V| - 1)}. \tag{3.3}$$

In an edge-partition we want the communities to be dense. In other words, for a partition $\mathscr{C} = \{C_0, C_1, C_2, ...C_N\}$ the subgraphs induced by each $V(C_i)$ should be dense according to 3.3. One possibility here is to take the average of this density for each community. But it might make more sense to consider the partition as a whole, and consider a sort of global density. The following is the density of the graph except we only count the pairs of nodes where both nodes are inside the same community. And we still count the pairs for each community independently, meaning the same pair can be counted several times if it is contained in several communities.

$$\rho = \frac{|E|}{pairs(\mathscr{C})}.$$

This also solves the problem of normalizing, it is guaranteed that $0 \leq \rho \leq 1$. The number of pairs inside communities must be at least $|E|$ since all the edges are inside communities and each edge represents a pair, so $\rho \leq 1$. And $\rho \geq 0$ since both factors are positive.

Unfortunately I don't know how to calculate $\mathbb{E}(\rho)$, so I cheat a little and calculate instead:

$$\frac{|E|}{\mathbb{E}\left(pairs(\mathscr{C})\right)}$$

This is not the same as $\mathbb{E}(\rho)$ but hopefully this is an adequate approximation. It tells us something about the average case and it does exclude the trivial cases, which was the purpose of the comparison in the first place (section 3.5). So the full expression of the Global Density inspired Modularity is:

$$GDM = \frac{|E|}{pairs(\mathscr{C})} - \frac{|E|}{\mathbb{E}(pairs(\mathscr{C}))} \tag{3.4}$$

Edge Modularity Unnormalized

Another way around the difficulty of normalizing, is to simply not normalize. This is not ideal, as the results for different graphs can't easily be compared. However this is easy to implement when 3.4 is already implemented. The Unnormalized Modularity is:

$$UM = \mathbb{E}(pairs(\mathscr{C})) - pairs(\mathscr{C}) \tag{3.5}$$

Calculating the Expectation of the Random Experiment

In the random experiment (section 3.5.4), we go through all the edges of the graph and assign a community to it. We will assign community C to a certain edge with probabilty $\frac{|C|}{|E|}$. To get the expectation we can loop through every pair of nodes and sum the probability. Let $l = |C|$ and $m = |E|$.

$$\mathbb{E}(pairs(\mathscr{C})) = \sum_{C \in \mathscr{C}} \sum_{u,v \in V} p^l_{u,v}$$

Where p^l_{uv} is the probability that u and v are both in $V(C)$, when the size of the community is l. If $uv \notin E$, then both u and v can have some other edge attached to them that is put into C. If $uv \in E$ then we have one more way that u and v can be put into $V(C)$: We put uv into C.

$$p^l_{uv} = \begin{cases} p^l_{k_u} p^l_{k_v}, & \text{if } uv \notin E \\ \frac{l}{m} + \left(1 - \frac{l}{m}\right) * p^l_{k_u - 1} p^l_{k_v - 1}, & \text{if } uv \in E \end{cases} \tag{3.6}$$

$p^l_{k_u}$ is the probability that a node with degree k_u is in V(C), when C has size l.

$$p^l_{k_u} = 1 - (1 - \frac{l}{m})^{k_u}$$

Here $\frac{l}{m}$ is the probability that one specific edge attached to u is in c. $(1 - \frac{l}{m})^{k_u}$ is the probability that none of the edges attached to u is in C.

Ratio of Number of Pairs to Expectation

After testing the algorithm with GDM and UM, I decided to add a third option, since the results of the first two were not completely satisfactory, and because this is a measure that's easy to implement when the other two are implemented already. It is simply the ratio of the expectation to the number of pairs. This measure should also be maximized.

$$Q_3 = \frac{\mathbb{E}(pairs(\mathscr{C}))}{pairs(\mathscr{C})} \tag{3.7}$$

Chapter 4

Implementation and Results of Minimizing Pairs

4.1 Implementation

I have tried to follow the implementation of Louvain when implementing my method for edges, but there are some differences in the implementation.

4.1.1 No Suitable Definition of Contracted Graph

In the Louvain algorithm, after each stage, when no more improvements can be gained by moving a node to another community, the algorithm contracts the graph. This is not meaningful when the communities consists of edges. When nodes are aggregated in the Louvain algorithm, we simply set the endpoints of the edges to be the communities of the original endpoints instead of the nodes themselves, and we let each community represent a node (this is better explained in section 2.1). This way we end up with multiedges and self loops. The natural way that Louvain deals with multiedges is to replace them with one edge that has weight equal to the sum of the weights of the original edges. If we were to aggregate the edges, the problem would be different. If we merge some edges in the in the graph into one edge, it is not clear what the endpoints of that edge would be. I do not see a way contract edges in a meaningful way, so I have done this part of the algorithm differently.

The important effect of the aggregation in Louvain is that once a stage is complete, the communities that were created during that stage will never be split into different communities. For instance if a community $C = \{u, v, w\}$ were created during the first stage of the Louvain algorithm, then those three nodes are guaranteed to be in the same community at the end of the entire algorithm. In my algorithm I don't aggregate the graph, but I get the same effect. Each community at the end of a stage can be a union of communities from the beginning of the stage.

So when the Louvain algorithm would treat one node, and try to put it into different communities to see if there is an increase in modularity. This algorithm treats one community as a whole and tries to take the union between this community and other communities to check if there is an increase in edge-modularity.

To illustrate, let's consider an example run of the algorithm on a graph with edges $\{e_i \mid 0 \leq i \leq 9\}$:

- First stage:
  - Communities at the beginning:
    $C_0 = \{e_0\}, C_1 = \{e_1\}, C_2 = \{e_2\}, C_3 = \{e_3\}, C_4 = \{e_4\},$
    $C_5 = \{e_5\}, C_6 = \{e_6\}, C_7 = \{e_7\}, C_8 = \{e_8\}, C_9 = \{e_9\}$
  - Communities at the end:
    $C'_0 = C_0 \cup C_1 \cup C_2 = \{e_0, e_1, e_2\},$
    $C'_1 = C_3 \cup C_4 = \{e_3, e_4\},$
    $C'_2 = C_5 \cup C_6 \cup C_7 = \{e_5, e_6, e_7\},$
    $C'_3 = C_8 \cup C_9 = \{e_8, e_9\}$
- Second stage:
  - Communities at the beginning:
    C'_0, C'_1, C'_2, C'_3
  - Communities at the end:
    $C''_0 = C'_0 = \{e_0, e_1, e_2\},$
    $C''_1 = C'_1 = \{e_3, e_4\},$
    $C''_2 = C'_2 \cup C'_3 = \{e_5, e_6, e_7, e_8, e_9\},$

- Third stage:
  - Nothing happens, so the algorithm ends.
- Communities at the end of the algorithm:
  $\{e_0, e_1, e_2\}$ $\{e_3, e_4\}$ $\{e_5, e_6, e_7, e_8, e_9\}$

4.1.2 Moving Not Only to Neighbouring Communities

In each stage the Louvain algorithm attempts to put each node into the community of each of its neighbours to check if there is a gain in modularity. It does not have to check communities where that node doesn't have a neighbour, because if the node doesn't have a neighbour in the community, then it is guaranteed that there will be a decrease in the modularity. This is fortunate for two reasons. It makes the algorithm more efficient, if it had to check every community the running time of the algorithm would always be quadratic in the number of nodes (since at the beginning every node is in it's own community). But perhaps the more important reason this is fortunate is that it wouldn't make much sense to have a node in a community where it has no neighbours.

For edge-modularity I would like a similar property. There should not be a gain in edge-modularity by putting two communities together if they do not share a border. For instance if we start out with communities C_0, C_1 on one stage of the algorithm, and $V(C_0) \cap V(C_1) = 0$ we should **not** get an increase in edge-modularity by putting C_0 and C_1 together.

I attempted to prove mathematically that each of the three measures in 3.6 have this property, but I couldn't prove this. I hoped that when running the algorithm on the data, it would only put communities together if they share a border. Because I did not know whether the measures would have this property, the algorithm checks all the communities in the graph, not only it's neighbours. I hoped that the algorithm would never put communities together if they do not share a border. However, it turns out that this can happen for each of the three measures in 3.6.

4.1.3 Computing Expectation

In the Louvain algorithm, the expectation of the random experiment is computed in constant time using equation 2.3. I do not have a constant time way of calculating the expectation. The expectation is a sum of the probabilities p^l_{uv} for each pair in each community. Where p^l_{uv} (equation 3.6) is the probability that a pair of nodes uv are both inside the same community of size l (see section 3.6). p^l_{uv} only depends on the degrees of the two nodes k_u and k_v, and whether there is an edge between them. The way I have implemented this is by creating two tables, S and T, of size $M * M$ where M is the largest degree in the graph. S_{k_u,k_v} is the number of pairs uv graph where u has degree k_u and v has degree k_v. T_{k_u,k_v} is the number of **edges** uv in the graph where u has degree k_u and v has degree k_v. This way I can calculate equation 3.6 only for each pair of degrees instead of per pair of nodes.

A more memory efficient alternative to this table would be an $N * N$ table where N is the number of different degrees in the graph. Each row and column of this table would correspond to degrees that actually are in the graph. As we don't usually need all possible degrees in this table, it will be smaller than the $M*M$ table. This table could be a bottleneck for memory. The highest degree possible in a graph with n nodes is $n-1$. As an example the biggest graph in table 4.1 has about 23000 nodes. A graph with this many nodes could have max degree 22999, and, if each element in the $M * M$ table is stored as an int taking 4 bytes, the memory used will be about $4\frac{22999^2}{2} = 264MB$. Luckily this is not an issue for the computer I've used to test. The running time of the algorithm is already at least $O(m^2)$, so this should not have much of an impact on the running time either.

4.1.4 Complexity

At the beginning of the first stage, every edge is inside it's own community. And for every community, the algorithm checks how much the modularity would increase when merging with one of the other, maximum m, number of communities. Then the algorithm might need to merge two communities. The time it takes to check whether two communities should be merged is $O(m)$. The time it takes to actually do the merge is also $O(m)$. This means the worst case complexity of one stage is $O(m^3)$. The number of stages will never exceed m, because at every stage, the algorithm has to merge at least two communities together, or the algorithm stops, and it can at most merge all the edges into one community. This means that in the worst case the complexity of the algorithm is $O(m^4)$. However, it runs faster on typical data 4.5, where it tends to only need 2-4 stages.

4.2 Results With Three Different Edge-Modularities

I have run the algorithm on several complex networks of increasing size. The three proposed definitions of modularity from section 3.6 is used. All results are from testing on the same computer. The computer has the following processor: Intel(R) Xeon(R) CPU E7- 4850 @ 2.00GHz, and 256GB RAM. I have tested the algorithm with each of the three measures in section 3.6 on 12 different networks displayed in table 4.1.

Dataset	nodes	edges	max degree
karate	34	78	17
foodweb	183	2.4k	108
figeys	2.2k	6.4k	314
moreno	1.7k	9.1k	364
as2000	6.4k	12.6k	1500
GrQc	4.1k	13.4k	81
HepTh	8.6k	24.8k	65
jung-j	6.1k	50.3k	26133
jdk	6.4k	53.7k	32530
as-caida	26.4k	53.4k	2600
CondMat	21.3k	91.3k	107
cora	23.2k	89.2k	379

Table 4.1: Data used for testing

karate is a social network of a karate club that split into two factions after an argument. *foodweb* is made up of foodchains in an ecosystem. *figeys* describes interactions between proteins in humans. *moreno* is a network describing proteins. *as2000* is describes subgraphs of the internet called autonomous systems. *GrQc* describes collaborations between authors in the field of general relativity and quantum cosmology. *HepTh* is a collaboration network in the field high energy physics. *jdk* describes software dependencies of the JDK framework. *s-caida* represents autonomous systems of the internet. *CondMat* is a collaboration network between authors writing about condense matter physics. *cora* is a citation network.

4.2.1 Results of Algorithm using GDM

The results of my algorithm using GDM is shown in table 4.3. The algorithm produces high values of GDM, the measure is between -1 and 1, and the values obtained are all above 0.5. This suggests that the algorithm does a good job optimizing the measure. However, although the measure is high for all of the results, the communities are not what we would expect in a good partition. This is apparent from the number of communities obtained. For each run of the algorithm the number of communities are close to the number of edges in the graph. This means that most edges end up in a community by itself. As an example, consider the last run of the algorithm on the network *cora*, the number of communities we obtain are 83900, and the number of edges in the network is 89200. If each community contained only one edge we would be left with only $89200 - 83900 = 5300$ edges, meaning that at most 5300 communities can contain more than one edge (since we can distribute those 5300 edges between at most 5300 communities). In other words we are left with at least $83900 - 5300 = 78600$ communities with only one edge. This means at least $78600/83900 = 94\%$ of the communities contain only one edge, and yet the GDM score is as high as 0.688. Since GDM gives high values for edge-partitions that are not good, we can conclude that it is not a good measure.

Dataset	#edges	#stages	Time	#com	GDM	UM /1000	RM
karate	78	2	0	62	0.556	0.11	2.3
		2	0	59	0.563	0.12	2.5
		2	0	61	0.587	0.12	2.5
foodweb	2400	2	6	1890	0.681	21.29	7.8
		2	6	1910	0.660	19.66	7.1
		2	6	1950	0.667	16.79	6.4
figeys	6400	3	88	6110	0.564	15.83	3.0
		2	88	6040	0.636	28.67	4.6
		3	85	5950	0.564	17.69	3.2
moreno	9100	2	91	8170	0.640	82.63	7.7
		2	89	8010	0.638	90.98	8.2
		2	105	8160	0.687	78.16	7.8
as2000	12600	3	887	11700	0.590	40.20	3.6
		3	700	11800	0.606	46.38	3.9
		3	906	12200	0.614	35.44	3.4
GrQc	13400	2	180	11800	0.790	44.93	28.4
		2	140	11500	0.761	35.73	22.2
		2	165	11800	0.757	237.6	15.3
HepTh	24800	3	764	22600	0.774	328.0	12.1
		3	1061	22500	0.740	366.6	12.8
		3	930	22300	0.776	445.5	15.8
jung-j	50300	2	53011	45360	0.626	714.7	10.8
		2	46088	45070	0.587	657.8	9.5
		2	44775	45150	0.624	669.8	10.2
jdk	53700	2	52728	48900	0.635	802.4	11.4
		2	43827	48460	0.574	922.6	11.7
		2	49318	48510	0.610	812.8	11.1
as-caida	53400	3	22098	52190	0.699	273100	5.3
		3	22106	51930	0.729	346900	6.5
		3	22589	52080	0.722	389800	7.1
CondMat	91300	3	11115	83380	0.631	748700	7.0
		3	14684	83190	0.650	944600	8.6
		4	15760	84570	0.641	740800	7.0
cora	89200	3	13470	83700	0.682	619600	6.5
		4	15288	84440	0.656	640400	6.5
		3	14462	83900	0.688	665900	6.9

Table 4.3: The results of my algorithm using GDM. There are three runs for each dataset, and in each stage of the algorithm the communities are considered in a random order. #S is the number of stages. Time is the running time in seconds. #C is the number of communities in the result. GDM, UM, and RM are the scores of the result with those edge-modularities. UM is counted /1000 (first entry is 110).

4.2.2 Comparison of all Three Measures

Results of the algorithm using the two other measures, UM and RM, are displayed in table 4.5 together with the results when using the first measure GDM. Here the results are the average between three runs of the algorithm, where each run considers the edges in a random order.

We can see that for each measure, the algorithm terminates within 2-4 stages. Keep in mind that the algorithm terminates when it goes through one stage without making any changes to the partition. In other words if it terminates after one stage, it means that it keeps the initial partition where each node is placed in a community by itself. So unless no changes are made to the algorithm, 2 stages is the minimum we will see.

When it comes to the number of communities created by the algorithm, only the runs with UM displays a sensible amount. As discussed in section 4.2.1, most of the communities gained using GDM contain only one edge. It is a little better using RM, but the number of communities are still close to the number of edges in the graph. Using RM on the dataset *cora* we get 66550 communities, and the number of edges is 89200. Looking at the number of communities using UM however, none of the values seem unreasonable. They are all in the range between 3 and 17, depending on the graph this could be a sensible number. It is worth noting however that both the Louvain algorithm and the algorithm by Evans et. al. generally obtain more communities than this on the same data (see figure 4.7).

As discussed in section 4.2.1, when the algorithm uses GDM we obtain a high GDM-score but most of the communities contain only one edge. If we look at the GDM score when the algorithm uses UM or RM we can see that it's very low. For the dataset *cora* the GDM score is 0.675 when GDM was used in the algorithm. When RM was used it is 0.07, and when UM is used it is 0.000678. Since the two other measures aren't normalized, it's hard to say whether the algorithm obtains high values UM when it uses UM, and whether it obtains high values of RM when using RM. But assuming the algorithm does a good job at optimizing each measure, in other words it gains a high value in the measure it uses, this suggests that GDM is not at all measuring the same thing as the two other measures. This seems a bit surprising since they are all based on the same idea, the number of pairs in the communities, and they use the same random experiment for comparison. The difference between the measures is how they compare the number of pairs inside communities to the expected number in the random experiment. The low GDM scores we obtain when the algorithm is run with UM, seems to reinforce the idea that GDM gives a higher score for many tiny communities.

Let's take a look at the times in table 4.5. The time used by the algorithm does not only depend on the size of the graph. The networks *jung-j* and *jdk* take more time, for each measure, than the larger networks *as-caida*, *CondMat* and *cora*, even though the number of stages is not necessarily higher. These are the networks that have the highest maximum degree among the ones I've used to test my algorithm. The algorithm is most likely slower on these networks because of how I calculate expectation (see section 4.1.3). This can be improved in the implementation however (this is also mentioned in section 4.1.3).

		algorithm using GDM						algorithm using UM						algorithm using RM					
Dataset	#edges	#S	Time	#com	GDM score	UM score /1000	RM score	#S	Time	#com	GDM score	UM score /1000	RM score	#S	Time	#com	GDM score	UM score /1000	RM score
karate	78	2	0	61	0.569	0.1	2.4	2.7	0	5	0.220	0.60	3.4	3	0	17	0.414	0.43	4.0
foodweb	2400	2	6	1917	0.669	19.2	7.1	2	8	17	0.130	104	7.4	3	7	1132	0.452	56	12.4
figeys	6400	2.7	87	6033	0.588	20.7	3.6	3.3	104	10	0.009	1597	3.9	3.3	102	4455	0.243	235	11.3
moreno	9100	2	95	8113	0.655	83.9	7.9	3	172	9.3	0.012	2952	5.8	3	132	5900	0.250	546	17.0
as2000	12600	3	831	1187	0.603	40.7	3.6	2.7	4281	3.7	0.003	9645	2.3	3.3	1533	10046	0.239	623	13.6
GrQc	13400	2	162	11700	0.769	34.8	22.0	2.7	339	8	0.005	10783	5.7	2.7	214	9477	0.359	2237	62.3
HepTh	24800	3	918	22466	0.763	380.0	13.6	2.7	1865	7	0.002	43717	4.4	3.7	1067	17577	0.137	3948	23.7
jung-j	50300	2	47958	45193	0.612	680.8	10.7	3	119165	9.7	0.006	48660	7.9	3	89469	34270	0.223	8338	39.1
jdk	53700	2	48624	48623	0.606	845.9	11.4	3	138508	9.3	0.006	55870	8.4	3.3	105480	36483	0.225	10180	44.5
as-caida	53400	3	22264	52066	0.717	336.6	6.3	3	95889	3.3	0.001	161167	2.1	3.3	13549	44697	0.176	7560	26.8
CondMat	91300	3.3	13853	83713	0.641	811.3	7.5	2	29971	8.3	0.001	430167	8.4	4	19214	66297	0.087	32087	32.3
cora	89200	3.3	14406	84013	0.675	641.9	6.6	2.7	34997	8.3	0.001	414733	5.0	4	19787	66550	0.070	28410	24.1

Table 4.5: Results of my algorithm using the three different measures. The values are the average results taken from three runs of the algorithm where the communities are considered in a random order. #S is the number of stages. Time is the running time in seconds. #com is the number of communities in the result. GDM, UM, and RM are the scores of the result with those edge-modularities. UM is counted /1000 (first entry is 110) in all three columns.

4.2.3 Analysis with the Karate Club Data

The Zachary karate club network is a social network of the members of a karate club that split into two groups after an argument between two of its leaders. The nodes are members of the club, and the links are ties between the members after the club split.

From figure 4.1 we can see that when the algorithm uses GDM it produces a partition where most communities contain only one edge. This is what we expect from the observations in section 4.2.1 and 4.2.2. Figure 4.1 shows that RM produces many tiny communities. RM only produce one community that contains only one edge, but the communities are very small. Each of the three figures in 4.1 were made with the algorithm considering communities in a random order.

When the algorithm is run using UM, figure 4.1, it produces 5 communities, and the size of each community looks more sensible. This partition however, does not seem like the most intuitive way to divide the graph either. At first sight it looks like the blue and green communities should have been merged into one community in figure 4.1 (UM), since visually they are very close to each other in a dense part of the graph. However, if we look more closely, we can see that the two communities are pretty separate. The only two nodes with edges from both communities incident to them are nodes 0 and 33. So if we merged these two communities, each node from the green community, except 0 and 33, would make a new pair with each node of the blue community, except 0 and 33. The new community would probably not look as much like a clique as the two old communities, since the new community would consist of two dense parts that are connected by only two nodes.

The red community in figure 4.1 (UM) can also look a bit surprising. It is spread throughout the graph, sharing a border with each of the other communities. Intuitively this should be split between the yellow and green/blue community. The black community also looks a bit surprising, it includes the triangle between the nodes 25, 26, and 32, even though only one edge ($\{1, 32\}$) connects it to the rest of the community. The black community also includes a cycle between the nodes 1, 8, 4, and 13. At first glance it looks like this cycle should belong to the yellow community, and looking closer we can see that three of the four nodes (1, 8, and 4) already have an edge in the yellow community incident to them. This means that if these edges was placed in the yellow community, let's call it C_{yellow}, the yellow community would get 4 more edges, and $V(C_{yellow})$ would only increase by 1. While having these 4 edges in the black community, C_{black}, means that the black community has 3 more edges but because of those three edges $V(C_{black})$ is increased by 3 nodes.

A possible explanation for why the communities are created this way when we use UM, is that the size of the communities might affect the decision of whether or not we merge two communities. For instance the algorithm might have decided to put the edges $\{\{1,8\},\{8,4\},\{4,13\},\{13,1\}\}$ into the black community instead of the yellow community because the black community was smaller. In fact when studying the algorithm step by step as it is performed on the karate club data with the measure UM, it looks like the size is relevant when making a choice of which communities to merge.

Size Effect of UM

When the algorithm uses UM it does not produce tiny communities like it does with GDM or RM, but there also seems to be a limit to how big communities it produces. To test this effect I have run the algorithm on the karate club data, but with additional disjoint cliques with 13 nodes each. The idea is to make the graph bigger, and study what happens with the partition of the part of the graph that represents the karate club. When one disjoint clique is added to the network it is twice the since of the original network in terms of edges, since $\frac{13*12}{2} = 78$, the exact number of edges in the karate klub network. I added cliques one by one and each time a new clique was added, I ran the algorithm 10 times, considering communities in a random order. Then I stopped when all the edges from the original karate club network was put into one community. The results are shown in figures 4.2 to 4.7.

Unfortunately the algorithm tends to place some edges from a clique in the same community as edges from the part of the network with the karate club, even though they are disjoint. To remedy this I have run the algorithm with the modification that I only consider merging communities that share a border node. See section 4.1.2 for a discussion about this.

By adding one clique (figure 4.2), doubling the size of the graph, the partition we obtain consists of 4 communities instead of the 5 communities in 4.1 (UM). However, the communities are still spread throughout the graph more than we might expect. Then one more clique is added, providing a graph three times the size of the original. The algorithm creates a partition with only three communities 4.3. With three cliques added, the graph is 4 times

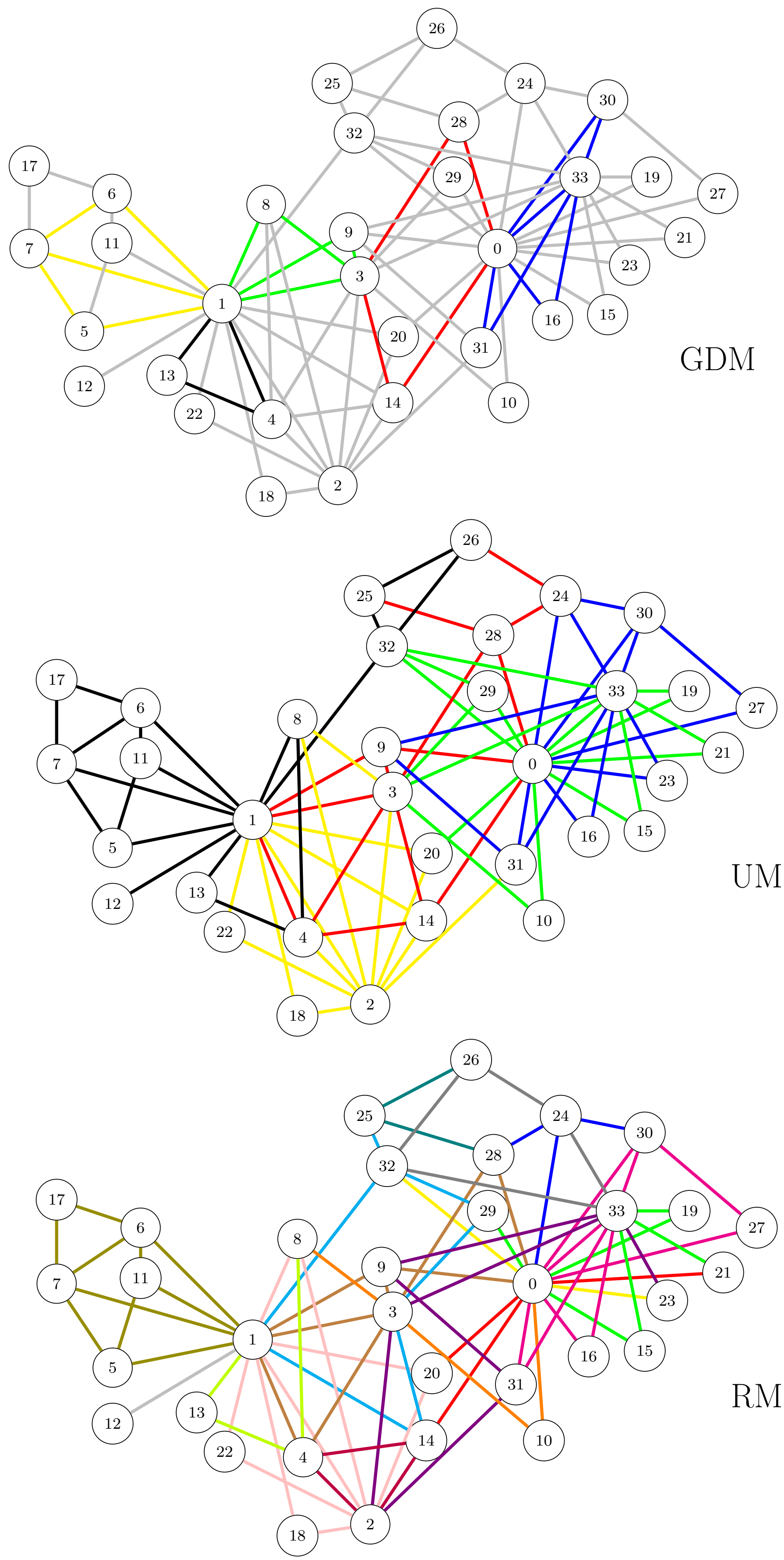

Figure 4.1: Results of my algorithm on the karate-club data using GDM, UM, and RM respectively. For GDM, the grey links represent edges that are alone in their community.

the size of the original, and the partition is split into just two communities. It's not hard to imagine that these two communities can represent how the members of the club split into two factions. With only three cliques added, the number of cliques has already decreased from 5 to 2. The communities we obtain when adding 4 and 5 cliques still has two communities. It only takes 6 cliques before the algorithm places every edge in one community. We can clearly see that the size of the network has an effect on the communities created. The larger the graph is, the larger communities are created.

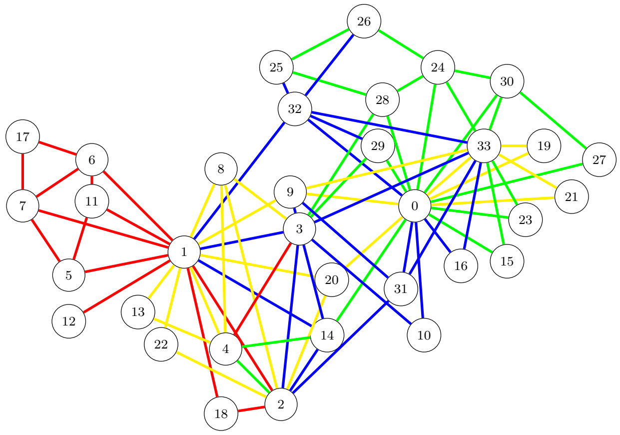

Figure 4.2: Result of algorithm using UM on the karate club data with 1 disjoint clique with 13 nodes added.

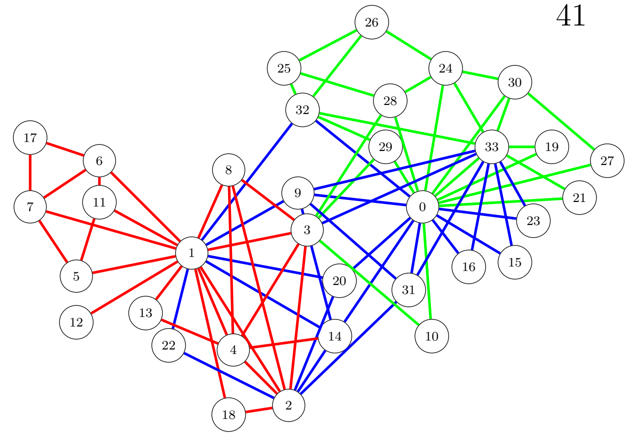

Figure 4.3: Result of algorithm using UM on the karate club data with 2 disjoint clique with 13 nodes added.

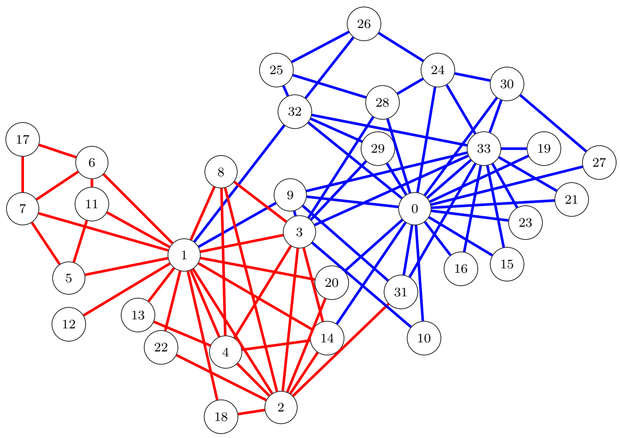

Figure 4.4: Result of algorithm using UM on the karate club data with 3 disjoint clique with 13 nodes added.

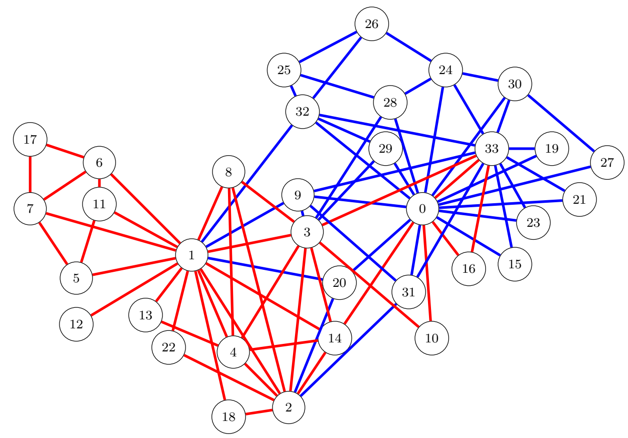

Figure 4.5: Result of algorithm using UM on the karate club data with 4 disjoint clique with 13 nodes added.

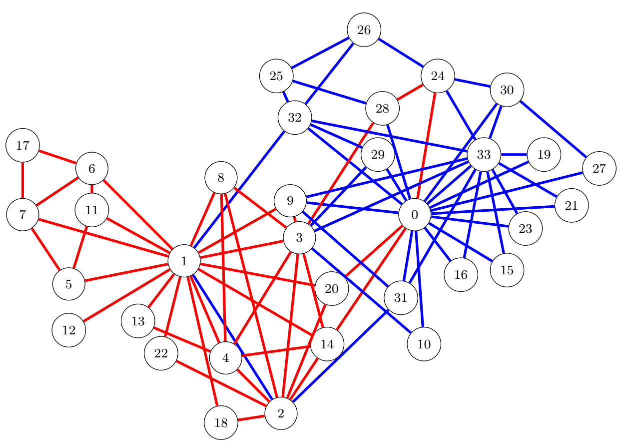

Figure 4.6: Result on the karate-club data with 5 disjoint clique with 13 nodes added. The edge-modularity used is GDM

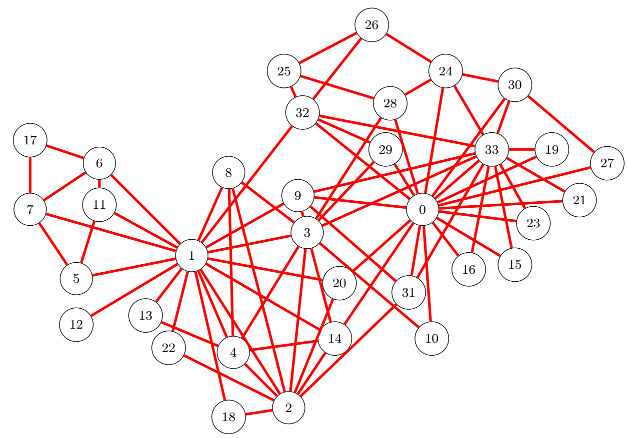

Figure 4.7: Result of algorithm using UM on the karate club data with 6 disjoint clique with 13 nodes added. Every edge is in the same community.

4.2.4 Results of the Louvain Algorithm and the Algorithm by Evans et. al.

I have included the results of my implementations of Louvain (section 2.1) and the two methods in [9] C and D (section 2.2). One interesting thing to note about the results (table 4.7) is that the datasets that required the most time with each of my measures are the same ones that require the most time here. It is the two graphs that have the highes maximum degree.

		results of louvain using C				results of louvain using D				results louvain			
Dataset	edges	#S	Time	#C	mod	#S	Time	#C	mod	#S	Time	#C	mod
karate	78	3	0	5	0.54	3	0	7	0.51	3	0	4	0.42
foodweb	2400	3	0.08	12	0.57	3	0.08	11	0.49	3	0	4	0.35
figeys	6400	5	0.12	32	0.87	5	0.39	32	0.59	5	0.01	13	0.46
moreno	9100	5	0.19	18	0.72	5	0.60	27	0.63	5	0.03	16	0.51
as2000	12600	4	0.64	18	0.67	5	1.66	45	0.70	5	0.30	28	0.62
GrQc	13400	5	0.25	32	0.81	5	0.30	45	0.86	6	0.03	42	0.84
HepTh	24800	5	0.53	50	0.80	6	0.76	60	0.79	5	0.09	50	0.76
jung-j	50300	5	12.33	22	0.71	6	26.55	44	0.64	5	0.06	14	0.48
jdk	53700	5	9.83	18	0.71	6	25.04	42	0.66	4	0.07	16	0.49
as-caida	53400	5	2.89	40	0.87	6	12.88	54	0.73	5	0.12	37	0.67
CondMat	91300	5	1.90	58	0.80	6	3.40	75	0.78	5	0.25	57	0.72
cora	89200	5	1.66	51	0.88	6	3.48	42	0.81	5	0.29	34	0.79

Table 4.7: Results of louvain with two of the graph-transformations in [9], C and D. And the results using louvain directly on the graph. #S is the number of stages. Time is the running time in seconds. #C is the number of communities in the result. mod is the modularity used in louvain.

Chapter 5

Discussion and Conclusion

5.1 Improving UM

Why Does UM Want to Merge Disconnected Communities?

Consider two communities C_1 and C_2, by disconnected I mean that $V(C_1) \cap V(C_2) = 0$. Even if C_1 and C_2 are disconnected, we can sometimes obtain a higher value of UM by merging C_1 and C_2 into one community. This is probably the main issue with UM, and a possible first step to improving the measure.

It seems like it does this because of how we compare to the random experiment in section 3.5.4. UM works by subtracting *pairs* from $\mathbb{E}(pairs)$, so UM is positive when $\mathbb{E}(pairs) > pairs$. Let's first take a look at the gain in *pairs*, Δ, when two small dense communities are merged.

Let's say C_1 and C_2 are two small dense edge communities. The increase in the number of pairs when we merge the communities, would be $\Delta = \frac{|V(C_1)|*|V(C_2)|}{2}$ (each node in $V(C_1)$ forms a new pair with each node in $V(C_2)$)

In the following discussion I will consider a typical outcome of the random experiment, instead of the expectation. This is just because it makes the argumentation easier, and a typical outcome will normally be close to the expected value. Let's compare this to a typical outcome of the random experiment (section 3.5.4). Let $C_1^{'}$ and $C_2^{'}$ be communities produced by the random experiment corresponding to C_1 and C_2 respectively. The increase of *pairs* in the random experiment when two communities are merged is

$$\Delta rand = \frac{(|V(C_1^{'})| - |V(C_1^{'}) \cap V(C_2^{'})|) * (|V(C_2^{'})| - |V(C_1^{'}) \cap V(C_2^{'})|)}{2}. \tag{5.1}$$

Each node in $V(C_1)$ that is not in $V(C_2)$ creates a new pair with each node in $V(C_2)$ that's not in $V(C_1)$. However $V(C_1^{'}) \cap V(C_2^{'})$ will likely be small, because $C_1^{'}$ and $C_2^{'}$ are small, and when choosing a small number of edges from a large graph, it is unlikely that many of those edges are incident to the same node. This means there will be little or no overlap of $V(C_1^{'})$ and $V(C_2^{'})$. In other words, if we disregard $|V(C_1^{'}) \cap V(C_2^{'})|$ in the expression of $\Delta rand$ it should not make a big difference. We end up with

$$\Delta rand \approx \frac{|V(C_1^{'})| * |V(C_2^{'})|}{2}.$$

The same expression as Δ, however it is unlikely that $C_1^{'}$ and $C_2^{'}$ will be dense, because we choose edges at random from the entire graph. Thus $V(C_1^{'})$ and $V(C_2^{'})$ will most likely contain more nodes than $V(C_1)$ and $V(C_2)$ respectively. Thus $\Delta rand$ will most likely be larger than Δ, meaning that there is an increase in UM if we merge the communities, even though the communities were originally unconnected.

As an example, let's say $|C_1| = |C_2| = 5$ are two communities with $V(C_1) \cap V(C_2) = 0$ and $|V(C_1)| = |V(C_2)| = 4$. Then it is likely that $V(C_1^{'}) = V(C_2^{'}) = 10$, if the graph is large. After the merge, *pairs* in the real partition increases by $\Delta = \frac{4*4}{2} = 8$. Meanwhile the increase in *pairs* in a typical outcome of the random experiment is $\Delta rand = \frac{10*10}{2} = 50$. When the expected increase of *pairs* in the random experiment is higher than the increase of *pairs* in the real partition, then UM will have a higher value after we merge.

It looks like a way to interpret this issue is that too much importance is given to $\mathbb{E}(pairs(\mathscr{C})$ when comparing it to $pairs(\mathscr{C})$, when dealing with small communities. One possible way to improve UM could be to remedy this issue in some way. For instance it might be possible to find some normalization factor, K, for the random part of UM.

$$UM_{improved} = \frac{\mathbb{E}(pairs(\mathscr{C}))}{K} - pairs(C)$$

The Size Effect of UM

The issue above seems to be because too much importance is placed on $\mathbb{E}(pairs(\mathscr{C}))$ when we deal with small communities. The size effect might also be a result of the comparison between $\mathbb{E}(pairs(\mathscr{C})$ and $pairs(\mathscr{C})$ being uneven. As mentioned above, when C is small, $V(C)$ in the random experiment is likely to be big. This is because each edge in C is likely to contribute two nodes to $V(C)$ in the random experiment. On the other hand if the community C is large compared to the graph, then it is much more likely that some edges contribute only one node, or no new nodes to $V(C)$. This is because when we choose an edge uv to be in C in the random experiment, it is likely that either $u \in V(c)$ or $v \in V(c)$. Meaning $V(C)$ will be smaller compared to C than it would be with a small community. In summary, it looks like $\mathbb{E}(pairs(\mathscr{C}))$ tends to be small when the community is big. Again this issue is about the comparison to the random experiment in UM, and could be improved by, for instance, some normalization factor.

5.2 Another Idea for Modularity of a Node Partition

An alternative modularity can be obtained by minimizing the number of edges across communities **and** the number of non-edges between nodes of $V(C_i)$. Let E_{out} be the edges that go across communities, and $E_{missing}$ be the set of non-edges between nodes inside $V(C_i)$).

$$E_{out} = \{uv |\ uv \in E,\ C(u) \neq C(v)\}$$

$$E_{missing} = \{uv |\ uv \notin E,\ C(u) = C(v)\}$$

We want a number between 0 and 1, so we need to normalize:

$$E_{out} \leq m$$

$$E_{missing} \leq \frac{n*(n-1)}{2} - m$$

The Measure we would like to minimize is

$$\frac{|E_{out}|}{2m} + \frac{|E_{missing}|}{n*(n-1) - 2m}.$$

Note that we use different normalization factors for each term. If we used the second factor $(1/(n(n-1)/2-m))$ for both the first term would be extremely small on sparse data compared to the second term. With this measure it might be unnecessary to compare to a random experiment. With Louvain modularity we need to compare to a random experiment because otherwise the optimal partition is just everything in one community. Here there is no such obvious problem.

5.3 Conclusion

A lot of research has been done on community detection in recent years. Among the methods for finding node partitions, the Louvain algorithm stands out as probably the most successful, and it works by optimizing a global measure of the quality of a partition, modularity [18]. In this thesis I have developed a model for link partitions that mimics the approach of the popular Louvain algorithm. The challenge of designing this method is to develop a version of modularity that works directly for edge partitions. I have provided several definitions of edge modularity. I implemented the new algorithm, and tested it on real data, using three different definitions of edge modularity. I also implemented the Louvain algorithm, and one other algorithm for edge partitions that uses the Louvain algorithm in it's approach. When testing the algorithm using the new edge modularities, one of the edge modularities, UM, seemed to provide more sensible communities than the others. In the end I discuss some ways in which UM could be improved.

Bibliography

[1] Cornell kddcup datasets. http://www.cs.cornell.edu/projects/kddcup/datasets.html. Accessed: 2019-03-30.

[2] EL ADNANI. A comprehensive literature review on community detection: Approaches and applications. *Procedia Computer Science*, 151:295–302, 2019.

[3] Yong-Yeol Ahn, James P. Bagrow, and Sune Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, Jun 2010. ISSN 1476-4687. doi: 10.1038/nature09182.
URL: `http://dx.doi.org/10.1038/nature09182`.

[4] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *nature*, 401(6749):130, 1999.

[5] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of community hierarchies in large networks. *CoRR*, abs/0803.0476, 2008.
URL: `http://arxiv.org/abs/0803.0476`.

[6] Sebastian Böcker and Jan Baumbach. Cluster editing. In *Conference on Computability in Europe*, pages 33–44. Springer, 2013.

[7] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. *On modularity-np-completeness and beyond*. Univ., Fak. für Informatik, Bibliothek, 2006.

[8] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, Sep. 1973. doi: 10.1147/rd.175.0420.

[9] T. S. Evans and Renaud Lambiotte. Edge partitions and overlapping communities in complex networks. *CoRR*, abs/0912.4389, 2009.
URL: `http://arxiv.org/abs/0912.4389`.

[10] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, Feb 2010. ISSN 0370-1573. doi: 10.1016/j.physrep.2009.11.002.
URL: `http://dx.doi.org/10.1016/j.physrep.2009.11.002`.

[11] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[12] Dongxiao He, Dayou Liu, Weixiong Zhang, Di Jin, and Bo Yang. Discovering link communities in complex networks by exploiting link dynamics. *CoRR*, abs/1303.4699, 2013.
URL: `http://arxiv.org/abs/1303.4699`.

[13] Dongxiao He, Di Jin, Zheng Chen, and Weixiong Zhang. Identification of hybrid node and link communities in complex networks. *Scientific reports*, 5:8638, 2015.

[14] Dongxiao He, Dayou Liu, Di Jin, and Weixiong Zhang. A stochastic model for detecting heterogeneous link communities in complex networks. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 130–136, 2015.
URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9372`.

[15] Zhenping Li, Xiang-Sun Zhang, Rui-Sheng Wang, Hongwei Liu, and Shihua Zhang. Discovering link communities in complex networks by an integer programming model and a genetic algorithm. *PloS one*, 8(12):e83739, 2013.

[16] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[17] Michael Molloy and Bruce Reed. A critical point for random graphs with a given degree sequence. *Random structures & algorithms*, 6(2-3):161–180, 1995.

[18] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2), Feb 2004. ISSN 1550-2376. doi: 10.1103/

physreve.69.026113.
URL: `http://dx.doi.org/10.1103/PhysRevE.69.026113`.

[19] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *nature*, 435(7043):814, 2005.

Appendix A

My Implementation of the Louvain Algorithm [5] and the algorithm by Evans et. al. [9]

Listing A.1: Source code of my implementation of the Louvain algorithm and the algorithm by Evans et. al. (both algorithms are in the same program).

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>
#include "rand.c"
#include "prelim.c"

char *IN_NAME = "./data/karate_right_numbers_converted";
bool RANDOM_ORDER = false;
bool RANDOM_TIEBREAKER = false;
double MIN_MOD_INCREASE = 0.001;
char *NODE2EDGE_FILENAME = "node2edge";
char *OUTPUT_FILENAME = "output";

/** type of graph to make
 * 0 - keep original
 * 1 - C (linegraph)
 * 2 - D (weighted linegraph)
 * 3 - E (weighted linegraph with self-loops)
 * 4 - E1 (weighted linegraph with self-loops)
**/
int TYPE = 0;

typedef struct Partition {
   /** number of nodes **/
   int n;

   /** sum of weights of edges inside each community **/
   double *inside;

   /** sum of weights of edges incident to some node in each community **/
   double *incident;
```

```

  /** mapping node i -> community **/
  int *node2comm;

} Partition;

typedef struct Edge {
  int dest;
  int origin;
  double weight;
} Edge;

Edge *node2edge;

typedef struct wgraph {
  int n;
  int m;
  Edge **links;
  int *degrees;
  double w;
  double *weighted_degrees;
  double *self_loops;
} wgraph;

/** assumes contiguous allocation of links! **/
void free_wgraph(wgraph *g) {
  free(g->degrees);
  free(g->weighted_degrees);
  free(g->self_loops);
  free(g->links[0]);
  free(g->links);
}

int *rand_perm(int n){
  int *perm;
  int i, tmp, j;
  if( (perm=(int *)malloc(n*sizeof(int))) == NULL )
    printf("random_perm: malloc() error");
  for (i=n-1;i>=0;i--)
    perm[i] = i;
  for (i=n-1;i>=0;i--){
    j = random()%(i+1);
    tmp = perm[i];
    perm[i] = perm[j];
    perm[j] = tmp;
  }
  return(perm);
}

/** makes weighted version of g in wg **/
void make_weighted(graph *g, wgraph *wg) {
  /* links */
  Edge **adj = malloc(g->n * sizeof *adj);
  adj[0] = malloc(g->m * 2 * sizeof **adj);
  for (int i = 1; i < g->n; i++) {
     adj[i] = adj[i-1] + g->degrees[i-1];
  }
  for (int i = 0; i < 2*g->m; i++) {
     Edge e;
     e.dest = g->links[0][i];
     e.weight = 1;
     adj[0][i] = e;
  }
```

```c

    /* weighted degree */
    double *weighted_degrees = malloc(g->n * sizeof *weighted_degrees);
    for (int i = 0; i < g->n; i++) {
       weighted_degrees[i] = g->degrees[i];
    }

    wg->n = g->n;
    wg->m = g->m;
    wg->degrees = g->degrees;
    wg->links = adj;
    wg->w = 2*g->m;
    wg->weighted_degrees = weighted_degrees;
    wg->self_loops = calloc(g->n, sizeof *wg->self_loops);
}

int** sort_adj_list(graph *g, int *half_degs) {
    /* allocate memory for new adjacency list */
    int **adj = (int**) calloc(g->n,sizeof(int*));
    adj[0] = (int*) calloc(g->m, sizeof(int));
    for (int i = 1; i < g->n; i++) {
       adj[i] = adj[i-1] + half_degs[i-1];
    }

    int *indices = (int*) calloc(g->n, sizeof(int));

    for (int u = 0; u < g->n; u++) {
       for (int j = 0; j < g->degrees[u]; j++) {
          int v = g->links[u][j];
          if (u < v) continue;
          adj[v][indices[v]++] = u;
       }
    }
    free(indices);
    return adj;
}

int* get_half_degs(graph *g) {
    int *degs = malloc(g->n * sizeof *degs);
    for (int i = 0; i < g->n; i++) {
       degs[i] = 0;
    }
    for (int u = 0; u < g->n; u++) {
       for (int j = 0; j < g->degrees[u]; j++) {
          int v = g->links[u][j];
          if (u < v) degs[u]++;
       }
    }
    return degs;
}

int* get_line_indices(graph *g, int *half_degs) {
    /* index of first edge connected to node u.
     * where edges (u,v) are only counted if u < v */
    int *edge_indices = (int*) malloc(g->n*sizeof(int));
    edge_indices[0] = 0;
    for (int u = 1; u < g->n; u++) {
       edge_indices[u] = edge_indices[u-1] + half_degs[u-1];
    }
    return edge_indices;
}

```

```c
int* get_line_degrees(graph *g, int **adj_sorted, int *edge_indices, int
    ↪ *half_degrees) {
   int *line_degrees = (int*) malloc(g->m*sizeof(int));
   for (int u = 0; u < g->n; u++) {
      for (int j = 0; j < half_degrees[u]; j++) {
         int v = adj_sorted[u][j];
         if (u >= v) continue;
         int index_uv = edge_indices[u] + j;
         line_degrees[index_uv] = g->degrees[u] + g->degrees[v] - 2;
      }
   }
   return line_degrees;
}

Edge** get_line_adj(graph *g, int *half_degs, int line_m, int
    ↪ *line_degrees) {
   bool use_self_loops = (TYPE == 3 || TYPE == 4);
   if (use_self_loops) {
      for (int i = 0; i < g->m; i++) {
         line_degrees[i]++;
      }
   }

   Edge **line_adj = malloc(g->m * sizeof *line_adj);
   if (use_self_loops) line_adj[0] = malloc( (line_m * 2 + g->m) * sizeof
    ↪ **line_adj);
   else line_adj[0] = malloc(line_m * 2 * sizeof **line_adj);
   for (int i = 1; i < g->m; i++) {
      line_adj[i] = line_adj[i-1] + line_degrees[i-1];
   }

   FILE *translation_file = fopen(NODE2EDGE_FILENAME, "w");
   node2edge = malloc(g->m * sizeof *node2edge);

   int *line_adj_indices = calloc(g->m, sizeof *line_adj_indices);
   int *edge_indices = get_line_indices(g, half_degs);
   int *not_added_twice = (int*) calloc(g->n, sizeof(int));
   int *edges_to_add = (int*) malloc(g->m*sizeof(int));
   int *self_loops = calloc(g->m, sizeof *self_loops);
   int num_edges_to_add = 0;
   for (int u = 0; u < g->n; u++) {
      num_edges_to_add = 0;
      int u_adj_index = 0;
      /* loop through neighbour edges (u,v) */
      for (int k = 0; k < g->degrees[u]; k++) {
         int v = g->links[u][k];
         int index_uv;
         if (u < v) index_uv = edge_indices[u] + u_adj_index++;
         else index_uv = edge_indices[v] + not_added_twice[v]++;
         edges_to_add[num_edges_to_add++] = index_uv;

         /* Translation back to edges */
         fprintf(translation_file, "%d %d %d\n", index_uv, u, v);
         Edge e;
         e.origin = u;
         e.dest = v;
         node2edge[index_uv] = e;

         /* add a self_loop */
         if (TYPE != 3 && TYPE != 4) continue;
         if (u < v) {
            Edge self_loop = {
               .dest = index_uv,
```

```
                .weight = 1./g->degrees[u] + 1./g->degrees[v],
            };
            line_adj[index_uv][line_adj_indices[index_uv]++] = self_loop;
            self_loops[index_uv] = self_loop.weight;
         }
      }
      /* create a link between each pair of neighbouring edges edges */
      for (int p = 0; p < num_edges_to_add; p++) {
         int e1 = edges_to_add[p];
         for (int q = 0; q < num_edges_to_add; q++) {
            if (p == q) continue;
            int e2 = edges_to_add[q];
            Edge e;
            e.dest = e2;
            if (TYPE == 1) e.weight = 1.0;
            else if (TYPE == 2) e.weight = 1.0/(g->degrees[u] -1);
            else if (TYPE == 3 || TYPE == 4) e.weight = 1.0/g->degrees[u];
            line_adj[e1][line_adj_indices[e1]++] = e;
         }
      }
   }
   free(line_adj_indices);
   free(edge_indices);
   free(not_added_twice);
   free(edges_to_add);
   fclose(translation_file);
   return line_adj;
}

int compare_edge( const void* a, const void* b)
{
     Edge edge_a = * ( (Edge*) a );
     Edge edge_b = * ( (Edge*) b );

     if ( edge_a.dest == edge_b.dest ) return 0;
     else if ( edge_a.dest < edge_b.dest ) return -1;
     else return 1;
}

/** Create E1 by E*E - E.
 *  g -> graph corresponding to E
**/
void create_E1(wgraph *g) {
   /* create adjacency matrix of E */
   double **E = malloc(g->n * sizeof *E);
   for (int i = 0; i < g->n; i++) {
      E[i] = malloc(g->n * sizeof **E);
   }
   /* init etries to -1 */
   for (int i = 0; i < g->n; i++) {
      for (int j = 0; j < g->n; j++) {
         E[i][j] = -1;
      }
   }

   /* fill table */
   for (int u = 0; u < g->n; u++) {
      for (int j = 0; j < g->degrees[u]; j++) {
         Edge e = g->links[u][j];
         E[u][e.dest] = e.weight;
      }
   }

```

```
    /* table with non-zero entries in E1 */
    bool **non_zero = malloc(g->n * sizeof *non_zero);
    for (int i = 0; i < g->n; i++) {
       non_zero[i] = malloc(g->n * sizeof **non_zero);
    }
    for (int i = 0; i < g->n; i++) {
       for (int j = 0; j < g->degrees[i]; j++) {
          non_zero[i][j] = false;
       }
    }

    /* allocate memory for result matrix */
    double **E1 = malloc(g->n * sizeof *E1);
    for (int i = 0; i < g->n; i++) {
       E1[i] = malloc(g->n * sizeof **E1);
    }
    for (int i = 0; i < g->n; i++) {
       for (int j = 0; j < g->n; j++) {
          E1[i][j] = 0;
       }
    }

    FILE *out = fopen("temp_debug", "w");

    /* do the math */
    /* E1 = E * E */
    for (int i = 0; i < g->n; i++) {
       for (int j = 0; j < g->n; j++) {
          for (int k = 0; k < g->n; k++) {
              if (E[i][k] > -1 && E[k][j] > -1) {
                 E1[i][j] += E[i][k] * E[k][j];
                 non_zero[i][j] = true;
             }
          }
       }
    }
    fclose(out);

    /* E1 = E1 - E */
    for (int i = 0; i < g->n; i++) {
       for (int j = 0; j < g->n; j++) {
          if (E[i][j] > -1) {
             E1[i][j] -= E[i][j];
             non_zero[i][j] = true;
          }
       }
    }

    /* create adj-list from matrix */
    /* degrees */
    int m = 0;
    double w = 0;
    int *degrees = malloc(g->n * sizeof *degrees);
    double *weighted_degrees = malloc(g->n * sizeof *weighted_degrees);
    int l = 0;
    for (int i = 0; i < g->n; i++) {
       degrees[i] = 0;
       weighted_degrees[i] = 0;
       for (int j = 0; j < g->n; j++) {
          if (non_zero[i][j]) {
             degrees[i]++;
             weighted_degrees[i] += E1[i][j];
             if (i <= j) l++;
```

```
            }
        }
        weighted_degrees[i] += E1[i][i]; //count self_loop twice
        m += degrees[i];
        w += weighted_degrees[i];
        w += E1[i][i];
    }
    // every edge exept self-loops are counted twice
    m += g->n;
    m /= 2;

    /* adj-list */
    Edge **adj = malloc(g->n * sizeof *adj);
    adj[0] = malloc((g->m * 2 + g->n) * sizeof **adj);
    for (int i = 1; i < g->n; i++) {
        adj[i] = adj[i-1] + degrees[i-1];
    }
    for (int i = 0; i < g->n; i++) {
        int k = 0;
        for (int j = 0; j < g->n; j++) {
            if (non_zero[i][j]) {
                Edge e;
                e.dest = j;
                e.weight = E1[i][j];
                adj[i][k++] = e;
            }
        }
        if (k > degrees[i]) report_error("\ndegree incoherence");
    }

    /* self-loops */
    double *self_loops = malloc(g->n * sizeof *self_loops);
    for (int i = 0; i < g->n; i++) {
        if (non_zero[i][i]) {
            self_loops[i] = E1[i][i];
        } else {
            report_error("\nself loop was zero");
        }
    }

    free(g->links[0]);
    free(g->links);
    free(g->degrees);
    free(g->weighted_degrees);
    free(g->self_loops);

    g->m = m;
    g->w = w;
    g->degrees = degrees;
    g->weighted_degrees = weighted_degrees;
    g->self_loops = self_loops;
    g->links = adj;
}

void make_linegraph(graph *g, wgraph *linegraph) {
    /* n */
    int line_n = g->m;

    /* m */
    int line_m = 0;
    for (int i = 0; i < g->n; i++) {
        line_m += (g->degrees[i] - 1) * g->degrees[i];
    }
```

```
    line_m /= 2;
    if (TYPE == 3 || TYPE == 4) line_m += line_n; // account for self-loops

    /* degrees */
    int *half_degs = get_half_degs(g);
    int *edge_indices = get_line_indices(g, half_degs);
    int **adj_sorted = sort_adj_list(g, half_degs);
    int *line_degrees = get_line_degrees(g, adj_sorted, edge_indices,
       ↪ half_degs);

    /* links */
    Edge **line_links = get_line_adj(g, half_degs, line_m, line_degrees);

    /* self loops */
    double *self_loops;
    if (TYPE == 3 || TYPE == 4) {
       self_loops = malloc(line_n * sizeof *self_loops);
       for (int i = 0; i < line_n; i++) {
          for (int j = 0; j < line_degrees[i]; j++) {
             if (line_links[i][j].dest == i)
                self_loops[i] = line_links[i][j].weight;
          }
       }
    } else {
       self_loops = calloc(line_n, sizeof self_loops);
    }
    /* weighted degrees */
    double *line_weighted_degrees = malloc(line_n * sizeof
       ↪ *line_weighted_degrees);
    if (TYPE == 1) {
       for (int i = 0; i < line_n; i++) {
          line_weighted_degrees[i] = line_degrees[i];
       }
    } else if (TYPE == 2) {
       for (int u = 0; u < line_n; u++) {
          line_weighted_degrees[u] = 0;
          for (int j = 0; j < line_degrees[u]; j++) {
             line_weighted_degrees[u] += line_links[u][j].weight;
          }
       }
    } else if (TYPE == 3 || TYPE == 4) {
       for (int i = 0; i < line_n; i++) {
          line_weighted_degrees[i] = 2;
          line_weighted_degrees[i] += self_loops[i];
       }
    }

    double line_w = 0;
    if (TYPE == 1) line_w = 2*line_m;
    if (TYPE == 2) {
       for (int i = 0; i < line_n; i++) {
          line_w += line_weighted_degrees[i];
       }
    }
    if (TYPE == 3 || TYPE == 4) {
       line_w = 2*line_n;
       for (int i = 0; i < line_n; i++) {
          line_w += self_loops[i];
       }
    }

    linegraph->self_loops = self_loops;
    linegraph->n = line_n;
```

```
   linegraph->m = line_m;
   linegraph->links = line_links;
   linegraph->degrees = line_degrees;
   linegraph->w = line_w;
   linegraph->weighted_degrees = line_weighted_degrees;

   if (TYPE == 4) create_E1(linegraph);

   free(edge_indices);
   free(adj_sorted);
   free(half_degs);
}

/* --------- Louvain ------------ */

long double modularity(wgraph *g, Partition *partition){
   bool *visited = (bool*) malloc((g->n)*sizeof(bool));
   for (int i = 0; i < g->n; i++) {
      visited[i] = false;
   }
   long double q = 0;
   long double w = (long double) g->w;
   for (int i = 0; i < g->n; i++) {
      int c = partition->node2comm[i];
      if (visited[c]) continue;
      visited[c] = true;

      q += 2*partition->inside[c];
      q -= ((partition->inside[c] + partition->incident[c])
         *  (partition->inside[c] + partition->incident[c])) / w;
   }
   q /= w;
   free(visited);
   return q;
}

long double modularity_gain(wgraph *g, Partition *partition, int node,
   ↪ int c, double k_in) {
   long double tot = (long double) partition->incident[c] +
      ↪ partition->inside[c];
   long double k = (long double) g->weighted_degrees[node];
   long double w = (long double) g->w;
   long double gain = (2*((long double)k_in) - 2*tot*k/w) / w;
   return gain;
}

void insert(int u, int c, Partition *partition, wgraph *g, double k_in_c)
   ↪ {
   partition->inside[c] += k_in_c + g->self_loops[u];

   partition->incident[c] += g->weighted_degrees[u];
   partition->incident[c] -= k_in_c;
   partition->incident[c] -= g->self_loops[u];
   partition->node2comm[u] = c;
}

void remove_node(wgraph *g, Partition *partition, int u, int c, double
   ↪ k_in_c) {
   partition->inside[c] -= k_in_c;
   partition->inside[c] -= g->self_loops[u];

   partition->incident[c] -= g->weighted_degrees[u];
   partition->incident[c] += k_in_c;
```

```
   partition->incident[c] += g->self_loops[u];
   partition->node2comm[u] = -1;
}

int* init_node2comm(wgraph *g) {
   /* initialize partition with one community per node */
   int *partition = (int*) malloc((g->n)*sizeof(int));
   for (int i = 0; i < (*g).n; i++) {
      partition[i] = i;
   }
   return partition;
}

void reset_neighbour_info(wgraph *g, int *node2comm, int u, bool *visited, double *k_in) {
   /* reset neighbours_in and visited */
   k_in[node2comm[u]] = 0;
   visited[u] = false;
   for (int i = 0; i < g->degrees[u]; i++) {
      int v = g->links[u][i].dest;
      int c = node2comm[v];
      visited[c] = false;
      k_in[c] = 0;
   }
   /* setting number of neighbours in each community for this node */
   for (int i = 0; i < g->degrees[u]; i++) {
      int v = g->links[u][i].dest;
      int c = node2comm[v];
      if (!(v == u)) { // does not count itself as a neighbour
         k_in[c] += g->links[u][i].weight;
      }
   }
}


bool should_visit(int u, int v, int *partition, bool *visited) {
   int c = partition[v];
   if (visited[c]) {
      return false;
   } else if (partition[u] == c) {
      return false;
   } else {
      visited[c] = true;
   }
   return true;
}

/**
 * returns a random community among <<best_communities>>
 * or -1 if max_gain is zero and the tiebreaker chooses the original community
**/
int tiebreak(int *best_communities, int n) {
   int k_max;
   int k;
   // choose random community among the best:
   k_max = n - 1;
   k = rand_lim(k_max);
   return best_communities[k];
}

/**
 * returns a community for node u among it's neighbouring communities
```

```c
**/
int best_assignment(int *best_communities, int num_best_comm) {
   int winner = -1;
   if (num_best_comm < 1) {
      return winner;
   }
   else if (!RANDOM_TIEBREAKER) {
      return best_communities[0]; //num_best_comm - 1];
   }
   /* tiebreaker */
   winner = tiebreak(best_communities, num_best_comm);
   return winner;
}

bool one_level(wgraph *g, Partition* partition) {
   bool *visited = (bool*) malloc((g->n)*sizeof(bool));
   double *k_in = (double*) malloc((g->n)*sizeof(double));
   int *best_communities = (int*) malloc((g->n)*sizeof(int));

   bool improvement = false;
   long double gain_this_round;
   long double max_gain;
   long double removal_gain;
   long double mod_incremental = modularity(g, partition);

   int round = 0;
   do { /* NEW ROUND */
      round++;
      gain_this_round = 0;

      int *node_perm;
      if (RANDOM_ORDER) {
         node_perm = rand_perm(g->n);
      } else {
         node_perm = malloc(g->n * sizeof *node_perm);
         for (int i = 0; i < g->n; i++) {
            node_perm[i] = i;
         }
      }

      for (int p = 0; p < g->n; p++) {
         int u = node_perm[p];     /* TREATING NODE u */
         if (g->degrees[u] == 1 && g->links[u][0].dest == u) return false; // only self as neighbour

         /* reset <<visited>> and <<neighbours_in>> for neighbourhood: */
         reset_neighbour_info(g, partition->node2comm, u, visited, k_in);

         /* remove node from old community */
         int old_community = partition->node2comm[u];
         remove_node(g, partition, u, old_community, k_in[old_community]);
         removal_gain = -modularity_gain(g, partition, u, old_community, k_in[old_community]);

         /* find all max gain communities among neighbours */
         max_gain = -3;
         int num_best_comm = 0; // number of communities with highest gain
         for (int i = 0; i < g->degrees[u]; i++) {
            int v = g->links[u][i].dest;
            int c = partition->node2comm[v];

            if (!should_visit(u, v, partition->node2comm, visited)) continue;
```

```c

                long double gain = removal_gain + modularity_gain(g, partition, u, c, k_in[c]);
                if (gain > max_gain) {
                    best_communities[0] = c;
                    num_best_comm = 1;
                    max_gain = gain;
                } else if (gain == max_gain) {
                    best_communities[num_best_comm++] = c;
                }
            }

            if (max_gain <= 0.000000) {
                // the node stays in it's old community
                best_communities[0] = old_community;
                num_best_comm = 1;
                max_gain = 0;
            }
            /* end find max gain communities */

            /* Assign node to community */
            int assign_to = best_assignment(best_communities, num_best_comm);
            insert(u, assign_to, partition, g, k_in[assign_to]);

            mod_incremental += max_gain;
            gain_this_round += max_gain;

        }
        if (gain_this_round > 0) improvement = true;
    } while (gain_this_round > MIN_MOD_INCREASE);
    free(visited);
    free(k_in);
    free(best_communities);
    return improvement;
}

void init_partition(wgraph *g, Partition *partition) {
    int *node2comm = init_node2comm(g);
    double *inside = calloc(g->n, sizeof *inside);
    double *incident = (double*) malloc((g->n)*sizeof(double));

    for (int i = 0; i < g->n; i++) {
        inside[i] = g->self_loops[i];
    }
    for (int i = 0; i < g->n; i++) {
        incident[i] = g->weighted_degrees[i];
        incident[i] -= g->self_loops[i]; //we should not count self_loops twice here
    }
    partition->inside = inside;
    partition->incident = incident;
    partition->node2comm = node2comm;
    partition->n = g->n;
}


void read_command_line_args(int argc, char **argv) {
    for (int i=1; i<argc; i++){
        if ((strcmp(argv[i],"-i")==0) || (strcmp(argv[i],"--input")==0) ) {
            IN_NAME = argv[++i];
        }
    } for (int i=1; i<argc; i++){
        if ((strcmp(argv[i],"-o")==0) || (strcmp(argv[i],"--output")==0) ) {
```

```
            OUTPUT_FILENAME = argv[++i];
        }
    } for (int i=1; i<argc; i++){
        if ((strcmp(argv[i],"-r")==0) || (strcmp(argv[i],"--random")==0) ) {
            RANDOM_ORDER = true;
            RANDOM_TIEBREAKER = true;
        }
    } for (int i=1; i<argc; i++){
        if ((strcmp(argv[i],"-m")==0) || (strcmp(argv[i],"--minmod")==0) ) {
            sscanf(argv[++i], "%lf", &MIN_MOD_INCREASE);
        }
    } for (int i=1; i<argc; i++){
        if ((strcmp(argv[i],"-e")==0) ||
            ↪ (strcmp(argv[i],"--edge-partition")==0) ) {
            sscanf(argv[++i], "%d", &TYPE);
        }
    }
}

/** shift community numbering **/
void renumber_partition(Partition *partition) {
    /* old2new */
    int *old2new = malloc(partition->n * sizeof *old2new);
    for (int i = 0; i < partition->n; i++) {
        old2new[i] = -1;
    }
    int k = 0; // new index of community
    for (int i = 0; i < partition->n; i++) {
        int old_c = partition->node2comm[i];
        if (old2new[old_c] == -1) old2new[old_c] = k++;
    }

    double *new_inside = malloc(partition->n * sizeof new_inside);
    double *new_incident = malloc(partition->n * sizeof new_incident);
    for (int i = 0; i < partition->n; i++) {
        new_inside[i] = 0;
        new_incident[i] = 0;
    }
    for (int i = 0; i < partition->n; i++) {
        int old_comm_index = partition->node2comm[i];
        int new_comm_index = old2new[old_comm_index];
        partition->node2comm[i] = new_comm_index;
        new_inside[new_comm_index] = partition->inside[old_comm_index];
        new_incident[new_comm_index] = partition->incident[old_comm_index];
    }
    free(partition->inside);
    free(partition->incident);
    partition->inside = new_inside;
    partition->incident = new_incident;
}

int count_communities(Partition *partition) {
    int n = 0;
    for (int i = 0; i < partition->n; i++) {
        if (partition->incident[i] > 0) n++;
    }
    return n;
}

/* communities must be numbered 0, 1, ... */
int* get_comm_sizes(wgraph *g, Partition *partition, int num_comms) {
    int *comm_sizes = calloc(num_comms, sizeof *comm_sizes);
    for (int i = 0; i < g->n; i++) {
```

```
      comm_sizes[partition->node2comm[i]]++;
   }
   return comm_sizes;
}

int** get_comm2nodes(wgraph *g, Partition *partition, int *comm_sizes, int num_comms) {
   int *comm_indices = calloc(num_comms, sizeof *comm_indices);
   int **comm2nodes = malloc(num_comms * sizeof *comm2nodes);
   comm2nodes[0] = malloc(g->n * sizeof **comm2nodes);
   for (int i = 1; i < num_comms; i++) {
      comm2nodes[i] = comm2nodes[i-1] + comm_sizes[i-1];
   }
   for (int i = 0; i < g->n; i++) {
      int c = partition->node2comm[i];
      comm2nodes[c][comm_indices[c]++] = i;
   }
   for (int i = 0; i < num_comms; i++) {
      if (comm_indices[i] != comm_sizes[i])
         report_error("get_comm_sizes: incoherence with indices");
   }
   free(comm_indices);
   return comm2nodes;
}

int* get_degrees(wgraph *g, Partition *partition, int *comm_sizes, int **comm2nodes, int n) {
   int *neighb = calloc(n, sizeof *neighb);
   int *degrees = malloc(n * sizeof *degrees);
   for (int c = 0; c < n; c++) {
      int deg_c = 0;
      /* count neighb. in each comm. */
      for (int i = 0; i < comm_sizes[c]; i++) {
         int u = comm2nodes[c][i];
         for (int j = 0; j < g->degrees[u]; j++) {
            int v = g->links[u][j].dest;
            int c_v = partition->node2comm[v];
            neighb[c_v]++;
         }
      }
      /* find degree by counting each neighb. comm. only once */
      for (int i = 0; i < comm_sizes[c]; i++) {
         int u = comm2nodes[c][i];
         for (int j = 0; j < g->degrees[u]; j++) {
            int v = g->links[u][j].dest;
            int c_v = partition->node2comm[v];
            if (neighb[c_v] == 1) deg_c++;
            neighb[c_v]--;
         }
      }
      degrees[c] = deg_c;
   }
   free(neighb);
   return degrees;
}

Edge** get_adj(wgraph *g, Partition *partition, int *degrees, int *comm_sizes, int **comm2nodes, int n, int m) {
   Edge **adj = malloc(n * sizeof *adj);
   adj[0] = malloc(m * sizeof **adj);
   for (int i = 1; i < n; i++) {
      adj[i] = adj[i - 1] + degrees[i - 1];
   }
```

```
   int *neighb = calloc(n, sizeof *neighb); // num negihbours in each comm
   double *neighb_weight = calloc(n, sizeof *neighb_weight);
   for (int c = 0; c < n; c++) {
      int k = 0; // neighbour index
      /* loop through nodes in comm,
      and count number of edges to neighbour comms
      and total weight of those edges */
      for (int i = 0; i < comm_sizes[c]; i++) {
         int u = comm2nodes[c][i];
         for (int j = 0; j < g->degrees[u]; j++) {
            int v = g->links[u][j].dest;
            int c_v = partition->node2comm[v];
            if (c_v == c && u > v) continue; // count edges inside c only once
            neighb[c_v]++;
            neighb_weight[c_v] += g->links[u][j].weight;
         }
      }
      /* find degree by counting each neighb. comm. only once */
      for (int i = 0; i < comm_sizes[c]; i++) {
         int u = comm2nodes[c][i];
         for (int j = 0; j < g->degrees[u]; j++) {
            int v = g->links[u][j].dest;
            int c_v = partition->node2comm[v];
            if (c_v == c && u > v) continue;
            if (neighb[c_v] == 1) {
               Edge e;
               e.dest = c_v;
               e.weight = neighb_weight[c_v];
               adj[c][k++] = e;
               neighb_weight[c_v] = 0;
            }
            neighb[c_v]--;
         }
      }
   }
   free(neighb);
   return adj;
}

wgraph* next_stage(wgraph *g, Partition *partition) {
   wgraph *new_graph = malloc(sizeof *new_graph);

   int n = count_communities(partition);
   int *comm_sizes = get_comm_sizes(g, partition, n);
   int **comm2nodes = get_comm2nodes(g, partition, comm_sizes, n);
   int *degrees = get_degrees(g, partition, comm_sizes, comm2nodes, n);
   int m = 0;
   for (int i = 0; i < n; i++) {
      m += degrees[i];
   }
   Edge **adj = get_adj(g, partition, degrees, comm_sizes, comm2nodes, n, m);

   double *self_loops = malloc(n * sizeof *self_loops);
   for (int u = 0; u < n; u++) {
      for (int j = 0; j < degrees[u]; j++) {
         if (adj[u][j].dest == u) {
            self_loops[u] = adj[u][j].weight;
         }
      }
   }

```

```c
    double w = 0;
    double *weighted_degs = malloc(n * sizeof *weighted_degs);
    for (int u = 0; u < n; u++) {
        weighted_degs[u] = 0;
        for (int j = 0; j < degrees[u]; j++) {
            weighted_degs[u] += adj[u][j].weight;
            w += adj[u][j].weight;
        }
        weighted_degs[u] += self_loops[u];
        w += self_loops[u];
    }

    new_graph->n = n;
    new_graph->m = m;
    new_graph->w = w;
    new_graph->degrees = degrees;
    new_graph->links = adj;
    new_graph->weighted_degrees = weighted_degs;
    new_graph->self_loops = self_loops;
    return new_graph;
}

void update_actual_partition(Partition *actual_partition, Partition *new_partition) {
    /* a community in actual must have numbering corresponding to it's node in new */
    for (int i = 0; i < actual_partition->n; i++) {
        actual_partition->inside[i] = 0;
        actual_partition->incident[i] = 0;
    }
    for (int i = 0; i < actual_partition->n; i++) {
        int old_comm = actual_partition->node2comm[i];
        int new_comm = new_partition->node2comm[old_comm];
        actual_partition->node2comm[i] = new_comm;
        actual_partition->inside[new_comm] = new_partition->inside[new_comm];
        actual_partition->incident[new_comm] = new_partition->incident[new_comm];

    }
}

void free_partition(Partition *partition) {
    free(partition->inside);
    free(partition->incident);
    free(partition->node2comm);
}

/**
 * Perform the louvain algorithm on g.
 * Returns the partition
 * @param stages will be updated with number of stages the algorithm used
**/
Partition* louvain(wgraph *g, int *stages) {
    /* initialize partition */
    Partition *partition = malloc(sizeof *partition);
    init_partition(g, partition);

    /* <<partition>> is the partition in the graph we edit,
     * we need to remember the partition as it is in the original graph */
    Partition *actual_partition = malloc(sizeof *actual_partition);
    init_partition(g, actual_partition);

```

```c
    /* Perform steps of the algorithm until we get no more improvement */
    bool improvement;
    int stage = 0;
    wgraph *new_graph = malloc(sizeof *new_graph);
    do {
        improvement = one_level(g, partition);
        renumber_partition(partition);
        update_actual_partition(actual_partition, partition);
        new_graph = next_stage(g, partition);
        free_partition(partition);
        init_partition(new_graph, partition);
        free_wgraph(g);
        g = new_graph;
    } while (improvement);

    *stages = stage;
    return actual_partition;
}

void create_wgraph(wgraph *g, char *in_name) {
    /* create unweighted graph from file */
    FILE *in_file = fopen(in_name, "r");
    graph *raw_graph = graph_from_file(in_file);
    fclose(in_file);

    /* create weighted graph */
    if (TYPE == 0) {
        make_weighted(raw_graph, g);
    } else {
        make_linegraph(raw_graph, g);
    }
}

void output_partition(wgraph *g, Partition *partition, int stages, double elapsed_time, long double mod, FILE *outfile) {
    int num_comm = count_communities(partition);
    int *comm_sizes = get_comm_sizes(g, partition, num_comm);
    int **comm2nodes = get_comm2nodes(g, partition, comm_sizes, num_comm);

    fprintf(outfile, "stages: %d \n", stages);
    fprintf(outfile, "elapsed time: %f \n", elapsed_time);
    fprintf(outfile, "num_coms: %d \n", num_comm);
    fprintf(outfile, "mod: %Lf \n", mod);

    for (int c = 0; c < num_comm; c++) {
        for (int i = 0; i < comm_sizes[c]; i++) {
            int u = comm2nodes[c][i];
            if (TYPE == 0) {
                fprintf(outfile, "%d %d\n", c, u);
            } else {
                Edge e = node2edge[u];
                fprintf(outfile, "%d %d %d\n", c, e.dest, e.origin);
            }
        }
    }
}

int main(int argc, char **argv) {
    //srand((unsigned) 102458);
    srand(time(NULL));

    /* command line arguments */
    read_command_line_args(argc, argv);
```

```

    /* Create weighted graph */
    wgraph *g = malloc(sizeof *g);
    create_wgraph(g, IN_NAME);

    /* Run the algorithm */
    int stages = 0;
    clock_t start_at = clock();
    Partition *partition = louvain(g, &stages);
    double elapsed = ((double) (clock() - start_at)) / CLOCKS_PER_SEC;

    /* recreate original graph */
    wgraph *original_graph = malloc(sizeof *original_graph);
    create_wgraph(original_graph, IN_NAME);
    long double mod_final = modularity(original_graph, partition);

    /* output file */
    FILE *outfile = fopen(OUTPUT_FILENAME, "w");
    output_partition(g, partition, stages, elapsed, mod_final, outfile);
    fclose(outfile);
}
```

Appendix B

The implementation of my algorithm

Listing B.1: Source code of my algorithm

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include <math.h>
#include <assert.h>
#include <float.h>

#include "prelim.c"
#include "rand.c"

char *IN_NAME = "./data/karate_right_numbers_converted";
//double MIN_MOD_INCREASE = 0.0;
char *OUTPUT_FILENAME = "output";
char *HISTORY_FILENAME = NULL;
bool RANDOM = false;
bool ONLY_NEIGHBOURS = false;
int MEASURE = -1;

void read_command_line_args(int argc, char **argv) {
   int i;
   for (i=1; i<argc; i++){
      if ((strcmp(argv[i],"-i")==0) || (strcmp(argv[i],"--input")==0) ) {
         IN_NAME = argv[++i];
      }
   } for (i=1; i<argc; i++){
      if ((strcmp(argv[i],"-o")==0) || (strcmp(argv[i],"--output")==0) ) {
         OUTPUT_FILENAME = argv[++i];
      }
   } for (i=1; i<argc; i++){
      if ((strcmp(argv[i],"-h")==0) || (strcmp(argv[i],"--history")==0) )
         ↪ {
         HISTORY_FILENAME = argv[++i];
      }
   } for (i=1; i<argc; i++){
      if ((strcmp(argv[i],"-m")==0) || (strcmp(argv[i],"--measure")==0) )
         ↪ {
         sscanf(argv[++i], "%d", &MEASURE);
      }
```

```
    } for (i=1; i<argc; i++){
        if ((strcmp(argv[i],"-r")==0) || (strcmp(argv[i],"--random")==0) ) {
            RANDOM = true;
        }
    } for (i=1; i<argc; i++){
        if ((strcmp(argv[i],"-n")==0) || (strcmp(argv[i],"--neighbours")==0) ) {
            ONLY_NEIGHBOURS = true;
        }
    }

}

//////////////////////////
// BEGIN : DATA STRUCTURES
//////////////////////////

typedef struct LocalEdge {
    // origin node ID
    int ori;
    // local number among the neighbours
    int nei_num;
} LocalEdge;

typedef struct EdgeCommunities {
    //number of communities
    int k;
    // number of edges in each community (table of size k)
    int* nb_edge;
    // number of nodes in each community (table of size k)
    int* nb_node;
    // list of edges in each community (table of size k pointing to a table of size m)
    LocalEdge** edge_list;
    // list of nodes in each community (table of size k pointing to a table of size <= 2m)
    int** node_list;
    // mapping edge (u,i) -> community (table of size n pointing to a table of size 2m, same as "links" for a graph)
    int** edge_to_com;
} EdgeCommunities;

void free_EdgeCommunities(EdgeCommunities *com) {
    free(com->nb_edge);
    free(com->nb_node);
    free(com->edge_list[0]);
    free(com->edge_list);
    free(com->node_list[0]);
    free(com->node_list);
    free(com->edge_to_com[0]);
    free(com->edge_to_com);
}

typedef struct SuperPartition {
    //number of supersets
    int p;
    // minimum free ID for a super set
    int freeID;
    // number of edges in each super set (table of size k with only p (non-consecutive) indices that are valid)
    int* nb_edge;
    // number of nodes in each super set (table of size k with only p (non-consecutive) indices that are valid)
```

```c
   int* nb_node;
   // mapping community -> super set (table of size k)
   int* com_to_sset;
} SuperPartition;

void free_SuperPartition(SuperPartition *spart) {
   free(spart->nb_edge);
   free(spart->nb_node);
   free(spart->com_to_sset);
}
///////////////////////
// END : DATA STRUCTURES
///////////////////////

typedef struct Edge {
   int dest;
   int origin;
   double weight;
} Edge;

Edge *node2edge;

int *rand_perm(int n){
  int *perm;
  int i, tmp, j;
  if( (perm=(int *)malloc(n*sizeof(int))) == NULL )
    printf("random_perm: malloc() error");
  for (i=n-1;i>=0;i--)
    perm[i] = i;
  for (i=n-1;i>=0;i--){
    j = random()%(i+1);
    tmp = perm[i];
    perm[i] = perm[j];
    perm[j] = tmp;
  }
  return(perm);
}

void print_communities(const graph *g, EdgeCommunities * com, FILE* fout) {
   int i;
   int j;
    for (i=0; i<com->k; i++) {
      fprintf(fout,"edges=%d, nodes=%d\n", com->nb_edge[i], com->nb_node[i]);
      for (j=0; j<com->nb_edge[i]; j++) {
         fprintf(fout,"(%d,%d) ",com->edge_list[i][j].ori,g->links[com->edge_list[i][j].ori][com->edge
      }
      fprintf(fout,"\n");
    }

}

/////////////////////
// BEGIN : EXPECTATION
/////////////////////

int find_max_deg(const graph *g) {
   int max_deg = 0;
   int i;
   for (i = 0; i < g->n; i++) {
      if (g->degrees[i] > max_deg) {
```

```c
            max_deg = g->degrees[i];
        }
    }
    return max_deg;
}

/** table with how many couples of each degreee-combination there are in
    ↪ g**/
int **get_S(const graph *g, int max_deg) {
    int **S = calloc((max_deg + 1), sizeof *S);
    int i;
    int u;
    int v;
    for (i = 0; i <= max_deg; i++) {
        S[i] = calloc((max_deg + 1), sizeof **S);
    }
    for (u = 0; u < g->n; u++) {
        for (v = 0; v < g->n; v++) {
            if (u == v) continue;
            S[g->degrees[u]][g->degrees[v]] += 1;
        }
    }
    /* couples between equal degree are counted twice in the table */
    for (i = 0; i < max_deg + 1; i++) {
        S[i][i] /= 2;
    }
    return S;
}

/** table with how many edges of each degreee-combination there are in g
    ↪ **/
int **get_T(const graph *g, int max_deg) {
    int **T = calloc((max_deg + 1), sizeof *T);
    int i;
    int j;
    int u;
    int v;
    for (i = 0; i <= max_deg; i++) {
        T[i] = calloc((max_deg + 1), sizeof **T);
    }
    for (u = 0; u < g->n; u++) {
        for (j = 0; j < g->degrees[u]; j++) {
            v = g->links[u][j];
            T[g->degrees[u]][g->degrees[v]] += 1;
        }
    }
    for (i = 0; i < max_deg; i++) {
        T[i][i] /= 2;
    }
    return T;
}

/** Probability that u and v is in V(C) of a community of size l,
 * if there is an edge between u and v **/
long double Puv_edge(int l, int ku, int kv, int m) {
    // works correctly for:
    // calculated for hand Puv_edge(1, 1, 4, 6) = 0.16666666666666666
    // calculated for hand Puv_edge(1, 2, 4, 6) = 0.225180
    if (ku == 0 || kv == 0) return 0;
    if (l == m) return 1;
    long double Puv = 0;
    Puv += pow(1.0 - (long double) l/(long double)m, ku + kv - 1);
    Puv -= pow(1.0 - (long double) l/(long double)m, ku);
```

```
    Puv -= pow(1.0 - (long double) l/(long double)m, kv);
    Puv += 1.0;
    return Puv;
}

/** Probability that u and v is in V(C) of a community of size l,
 * if there is no edge between u and v **/
long double Puv_noedge(int l, int ku, int kv, int m) {
    // works correctly for:
    // calculated for hand Puv_noedge(1, 1, 4, 6) = 0.08629115226337448
    if (ku == 0 || kv == 0) return 0;
    if (l == m) return 1;
    long double Puv = 0;
    Puv += pow(1.0 - (long double) l/(long double)m, ku + kv);
    Puv -= pow(1.0 - (long double) l/(long double)m, ku);
    Puv -= pow(1.0 - (long double) l/(long double)m, kv);
    Puv += 1.0;
    return Puv;
}

long double expectation(int l, const graph *g, bool* calculated, long
    ↪ double* expectation_table) {
    if (calculated[l]) return expectation_table[l];
    int ku;
    int kv;
    int i;

    int max_deg = find_max_deg(g);
    int **S = get_S(g, max_deg);
    int **T = get_T(g, max_deg);

    long double expectation = 0;
    for (ku = 0; ku <= max_deg; ku++) {
        for (kv = 0; kv <= max_deg; kv++) {
            if (ku > kv) continue;
            long double edge = (long double) T[ku][kv];
            long double noedge = (long double) S[ku][kv] - edge;
            expectation += noedge * Puv_noedge(l, ku, kv, g->m);
            expectation += edge * Puv_edge(l, ku, kv, g->m);
        }
    }

    for (i = 0; i < max_deg; i++) {
        free(S[i]);
        free(T[i]);
    }
    free(S);
    free(T);

    expectation_table[l] = expectation;
    calculated[l] = true;
    return expectation;
}

/////////////////////
// END : EXPECTATION
/////////////////////

// create and initialise a super partition from a given partition into
    ↪ communities by putting each community alone in its super set
SuperPartition* init_superpart (const EdgeCommunities* com) {
    int c;
     SuperPartition* spart;
```

```
     if( (spart=(SuperPartition *)malloc(sizeof(SuperPartition))) == NULL )
        report_error("init_superpart: malloc() error");
    spart->p = com->k;
    spart->freeID = spart->p;

    if( (spart->nb_edge=(int*)malloc(com->k * sizeof(int))) == NULL )
        report_error("init_superpart: malloc() error");
    if( (spart->nb_node=(int*)malloc(com->k * sizeof(int))) == NULL )
        report_error("init_superpart: malloc() error");
     if( (spart->com_to_sset=(int*)malloc(com->k * sizeof(int))) == NULL )
        report_error("init_superpart: malloc() error");

    for (c = 0; c < com->k; c++) {
       spart->nb_edge[c] = com->nb_edge[c];
       spart->nb_node[c] = com->nb_node[c];
       spart->com_to_sset[c] = c;
    }
    return spart;
}

////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////   UPDATE_COMMUNITIES //////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
/// IN: spart, g
/// IN/OUT: com, visited_nodes (comes back to its initial value at the end of the procedure)
/// OUT:
////////////////////////////////////////////////////////////////////////////////////////
/// PRE-REQUISITE: all cells of visted nodes contain the value -1 and spart is a proper partition of the communities in com, which are communities of graph g
/// RESULT: update com by merging the communities belonging to the same part of the super partition spart
////////////////////////////////////////////////////////////////////////////////////////
void update_communities(EdgeCommunities* com, const SuperPartition* spart, const graph* g, int* visited_nodes) {

    int l;
    int i,j;
     int u,v;
     LocalEdge** new_edge_list;
     int* cur_edge;
     int* nb_com;
     int** com_list;
     int* cur_com_list;
     int** new_node_list;

     // build a table new_com of mapping from old communities to new comunity number from 0 to p-1
     // update com->nb_edge et com->nb_node (old values are lost)
     int* new_com;
     int* new_nb_edge;
     int* new_nb_node;
     int cur_com;

     if( (new_com=(int *)malloc(com->k*sizeof(int))) == NULL )
        report_error("update_communities: malloc() error");
     if( (new_nb_edge=(int *)malloc(spart->p*sizeof(int))) == NULL )
        report_error("update_communities: malloc() error");
     if( (new_nb_node=(int *)malloc(spart->p*sizeof(int))) == NULL )
        report_error("update_communities: malloc() error");

```

```c
    cur_com = 0;
    for (i=0; i<com->k; i++) {
        if (spart->nb_edge[i]!=-1) {
            new_com[i] = cur_com;
            new_nb_edge[cur_com]=spart->nb_edge[i];
            new_nb_node[cur_com]=spart->nb_node[i];
            cur_com++;
        }
        else
            new_com[i] = -1;
    }
    if (cur_com != (spart->p)) report_error("update_communities: incoherence with p");

    // update com->edge_to_com
    for (u=0; u<g->n; u++) {
        for (v=0; v<g->degrees[u]; v++) {
            com->edge_to_com[u][v] = new_com[spart->com_to_sset[com->edge_to_com[u][v]]];
        }
    }

    // update com->edge_list
    if( (new_edge_list=(LocalEdge**)malloc(spart->p*sizeof(LocalEdge*))) == NULL )
       report_error("update_communities: malloc() error");
    if( (new_edge_list[0]=(LocalEdge*)malloc(g->m*sizeof(LocalEdge))) == NULL )
       report_error("update_communities: malloc() error");
    for (i=1; i<spart->p; i++) {
        new_edge_list[i] = new_edge_list[i-1]+new_nb_edge[i-1];
    }
    if( (cur_edge=(int *)malloc(spart->p*sizeof(int))) == NULL )
       report_error("update_communities: malloc() error");
    for (i=0; i<spart->p; i++) cur_edge[i] = 0;

    for (i=0; i<com->k; i++) {
        for (j=0; j<com->nb_edge[i]; j++) {
            new_edge_list[new_com[spart->com_to_sset[i]]][cur_edge[new_com[spart->com_
        }
        cur_edge[new_com[spart->com_to_sset[i]]] += com->nb_edge[i];
    }

    for (i=0; i<spart->p; i++) {
        if (cur_edge[i] != new_nb_edge[i]) report_error("update_communities: incoherence in new_edge_list");
    }

    free(cur_edge);

    // update com->node_list
    if( (nb_com=(int*)malloc(spart->p*sizeof(int))) == NULL )
       report_error("update_communities: malloc() error");
    for (i=0; i<spart->p; i++) {
        nb_com[i]=0;
    }
    for (i=0; i<com->k; i++) {
        nb_com[new_com[spart->com_to_sset[i]]]++;
    }

    if( (com_list=(int**)malloc(spart->p*sizeof(int*))) == NULL )
       report_error("update_communities: malloc() error");
```

```
    if( (com_list[0]=(int*)malloc(com->k*sizeof(int))) == NULL )
        report_error("update_communities: malloc() error");
    for (i=1; i<spart->p; i++) {
        com_list[i] = com_list[i-1]+nb_com[i-1];
    }

    if( (cur_com_list=(int*)malloc(spart->p*sizeof(int))) == NULL )
        report_error("update_communities: malloc() error");
    for (i=0; i<spart->p; i++) {
        cur_com_list[i]=0;
    }

    for (i=0; i<com->k; i++) {
        com_list[new_com[spart->com_to_sset[i]]][cur_com_list[new_com[spart->com_to_ss
        cur_com_list[new_com[spart->com_to_sset[i]]]++;
    }
    for (i=0; i<spart->p; i++) {
        if (cur_com_list[i] != nb_com[i]) report_error("update_communities: incoherence in com_list");
    }
    free(cur_com_list);

    if( (new_node_list=(int**)malloc(spart->p*sizeof(int*))) == NULL )
        report_error("update_communities: malloc() error");
    if( (new_node_list[0]=(int*)malloc(2*g->m*sizeof(int))) == NULL )
        report_error("update_communities: malloc() error");
    for (i=1; i<spart->p; i++) {
        new_node_list[i] = new_node_list[i-1]+new_nb_node[i-1];
    }

    int cur_merge;
    for (i=0; i<spart->p; i++) {
        cur_merge = 0;
        for (j=0; j<nb_com[i]; j++) {
            for (l=0; l<com->nb_edge[com_list[i][j]]; l++) {
                u=com->edge_list[com_list[i][j]][l].ori;
                v=g->links[u][com->edge_list[com_list[i][j]][l].nei_num];
                if (visited_nodes[u] == -1) {
                    visited_nodes[u]=1;
                    new_node_list[i][cur_merge]=u;
                    cur_merge++;
                }
                if (visited_nodes[v] == -1) {
                    visited_nodes[v]=1;
                    new_node_list[i][cur_merge]=v;
                    cur_merge++;
                }
            }
        }
        if (cur_merge != new_nb_node[i]) report_error("update_communities: incoherence in node_list");
        //reset visited_nodes
        for (j=0; j<new_nb_node[i]; j++) {
            visited_nodes[new_node_list[i][j]] = -1;
        }
    }

    free(com->node_list[0]);
    free(com->node_list);
    com->node_list=new_node_list;

    // update com->k
```

```c
    com->k = spart->p;

    // update com->nb_edge and com->nb_node
    free(com->nb_edge);
    free(com->nb_node);
    com->nb_edge=new_nb_edge;
    com->nb_node=new_nb_node;

    //update com->edge_list
    free(com->edge_list[0]);
    free(com->edge_list);
    com->edge_list = new_edge_list;

    //free memory
    free(new_com);
    free(nb_com);
    free(com_list[0]);
    free(com_list);

}

////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////////   NODE_DIFF ////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
/// IN:
/// IN/OUT:
/// OUT:
////////////////////////////////////////////////////////////////////////////////
/// PRE-REQUISITE:
/// RESULT: for c a community not in super set sset, return the number of nodes of c that are not in sset
////////////////////////////////////////////////////////////////////////////////
int node_diff (const int c, const int sset, const EdgeCommunities* com, const SuperPartition* spart, const graph* g) {
    int diff = 0;

    int u;
    bool in_sset;
    int i,j;

    for (i = 0; i < com->nb_node[c] ; i++) {
      u = com->node_list[c][i];

      in_sset = false;
      j = 0;
      while ((j < g->degrees[u] ) && !in_sset) {
        if (spart->com_to_sset[com->edge_to_com[u][j]]==sset) in_sset = true;
        j++;
      }
      if (!in_sset) diff++;

   }
   return diff;
}

int count_couples(EdgeCommunities *com) {
   int couples = 0;
   int c;
   for (c = 0; c < com->k; c++) {
      couples +=  (com->nb_node[c] * (com->nb_node[c] - 1))/2;
   }
```

```
    return couples;
}

long double calculate_expectation(const graph *g, EdgeCommunities *com, bool *calculated, long double * expectation_table) {
    long double expect = 0;
    int c;
    for (c = 0; c < com->k; c++) {
        expect += expectation(com->nb_edge[c], g, calculated, expectation_table);
    }
    return expect;
}

// returns modularity of edge-partition, using the formula:
// Q = |E|*( 1/couples(partition) - 1/E(couples(partition)) )
long double edge_modularity(const graph *g, EdgeCommunities *partition, bool* calculated, long double* expectation_table) {
    int couples = count_couples(partition);
    long double q1 = ((long double) g->m) / ((long double) couples);

    long double expect = calculate_expectation(g, partition, calculated, expectation_table);
    long double q2 = ((long double) g->m) / expect;

     return q1 - q2;
}

// returns modularity of edge-partition, using the formula:
// Q = |E|*( couples(partition) - E(couples(partition)) )
long double edge_mod_minus(const graph *g, EdgeCommunities *partition, bool* calculated, long double* expectation_table) {
    int couples = count_couples(partition);
    long double expect = calculate_expectation(g, partition, calculated, expectation_table);
     return (expect - (long double)couples);
}

// returns modularity of edge-partition, using the formula:
// Q = |E|*( E(couples(partition))/couples(partition))
long double edge_mod_ratio(const graph *g, EdgeCommunities *partition, bool* calculated, long double* expectation_table) {
    int couples = count_couples(partition);
    long double expect = calculate_expectation(g, partition, calculated, expectation_table);
     return expect / couples;
}

// gain in modularity for putting community c into sset
long double edge_mod_gain(int c, int sset, int couples_in_spart, long double expect_spart, EdgeCommunities * com, SuperPartition *spart, const graph *g, bool* calculated, long double* expectation_table) {

    if (expect_spart <= 0.0)
        report_error("The expected number of couples in the partition must be positive.\n");
    if (couples_in_spart <= 0)
     report_error("the number of couples in the superpartition must be positive.\n");

    // COUPLES
    int diff_nodes = node_diff(c, sset, com, spart, g);
    int nodes_sset = spart->nb_node[sset];
```

```c
    int nodes_c = com->nb_node[c];

    int couples_after = couples_in_spart;
    couples_after -= (spart->nb_node[sset] * (spart->nb_node[sset] - 1)) / 2;
    couples_after -= (nodes_c * (nodes_c - 1)) / 2;
    couples_after += ((nodes_sset + diff_nodes) * (nodes_sset + diff_nodes - 1)) / 2;

    // EXPECTED COUPLES
    long double expectation_after = expect_spart;
    expectation_after -= expectation(com->nb_edge[c], g, calculated, expectation_table);
    expectation_after -= expectation(spart->nb_edge[sset], g, calculated, expectation_table);
    expectation_after += expectation(com->nb_edge[c] + spart->nb_edge[sset], g, calculated, expectation_table);

    // DELTA MODULARITY
    // modularity is mod after merge, minus mod before
    long double mod;
    if (MEASURE == 1) {
       // with first idea of modularity
       mod = (long double) g->m / ((long double) (couples_after)) - (long double) g->m / ((long double) (expectation_after));
       mod -= (long double) g->m / ((long double) couples_in_spart) - (long double) g->m / ((long double) expect_spart);
    } else if (MEASURE == 2) {
       // with second idea of modularity
       mod = ((long double) (couples_after)) - ((long double) (expectation_after));
       mod -= ((long double) couples_in_spart) - ((long double) expect_spart);
       mod = -mod;
    } else if (MEASURE == 3) {
       // with third idea of modularity
       mod = ((long double) (expectation_after)) / ((long double) (couples_after)) ;
       mod -= ((long double) expect_spart) / ((long double) couples_in_spart);
    } else report_error("measure must be given with -m option\n");

    return mod;
}

// Returns sorted adj-list
int** sort_adj_list(graph *g) {
    int i, u, j;
    //allocate memory for new adjacency list
    int **adj = (int**) calloc(g->n,sizeof(int*));
    adj[0] = (int*) calloc(2*g->m, sizeof(int));
    for (i = 1; i < g->n; i++) {
       adj[i] = adj[i-1] + g->degrees[i-1];
    }

    int *indices = (int*) calloc(g->n, sizeof(int));

    for (u = 0; u < g->n; u++) {
       for (j = 0; j < g->degrees[u]; j++) {
          int v = g->links[u][j];
          adj[v][indices[v]++] = u;
       }
    }
```

```
    free(indices);
    return adj;
}

void init_edge_communities(const graph *g, EdgeCommunities* partition) {
    int i;
     partition->k = g->m;

    //Allocate nb_edge, nb_node
    if( (partition->nb_edge=(int *)malloc(partition->k*sizeof(int))) == NULL )
        report_error("init_edge_communities: malloc() error");
     for (i = 0; i < partition->k; i++) {
         partition->nb_edge[i] = 1;
     }
    if( (partition->nb_node=(int *)malloc(partition->k*sizeof(int))) == NULL )
        report_error("init_edge_communities: malloc() error");
     for (i = 0; i < partition->k; i++) {
         partition->nb_node[i] = 2;
     }

    //Allocate edge_list
    if( (partition->edge_list=(LocalEdge **)malloc(partition->k*sizeof(LocalEdge*))) == NULL )
        report_error("init_edge_communities: malloc() error");
     if( (partition->edge_list[0]=(LocalEdge *)malloc(g->m * sizeof(LocalEdge))) == NULL )
        report_error("init_edge_communities: malloc() error");
     for (i = 1; i < partition->k; i++) {
         partition->edge_list[i] = partition->edge_list[i-1] + 1;
     }

    //Allocate node_list
    if( (partition->node_list=(int **)malloc(partition->k*sizeof(int*))) == NULL )
        report_error("init_edge_communities: malloc() error");
     if( (partition->node_list[0]=(int *)malloc(2*g->m * sizeof(int))) == NULL )
        report_error("init_edge_communities: malloc() error");
     for (i = 1; i < partition->k; i++) {
         partition->node_list[i] = partition->node_list[i-1] + 2;
     }

    //Allocate edge_to_com
    if( (partition->edge_to_com=(int **)malloc(g->n*sizeof(int*))) == NULL )
        report_error("init_edge_communities: malloc() error");
     if( (partition->edge_to_com[0]=(int *)malloc(2*g->m * sizeof(int))) == NULL )
        report_error("init_edge_communities: malloc() error");
     for (i = 1; i < g->n; i++) {
         partition->edge_to_com[i] = partition->edge_to_com[i-1] + g->degrees[i-1];
     }

    // Fill edge_list, node_list, and edge_to_com
     //int **sorted_adj = sort_adj_list(g);
     int* indices;
     if( (indices=(int *)malloc(g->n*sizeof(int))) == NULL )
        report_error("init_edge_communities: malloc() error");
     for (i = 0; i < g->n; i++) indices[i]=0;

```

```
    int u,j,l;
    int com = 0;
     for (u = 0; u < g->n; u++) {
         for (j = 0; j < g->degrees[u]; j++) {
          int v = g->links[u][j];
          if (u < v) {

              // fill edge_list
           partition->edge_list[com][0].ori = u;
           partition->edge_list[com][0].nei_num = j;

           // fill node_list
           partition->node_list[com][0] = u;
           partition->node_list[com][1] = v;

           // fill edge_to_com
           partition->edge_to_com[u][j] = com;

           com++;
        }
        else {
         l=0;
         while (g->links[v][l]!=u) l++;
         partition->edge_to_com[u][j] = partition->edge_to_com[v][l];
        }
         }
     }

    free(indices);
}

// Remove c from it's superset and put it in a new superset freeID. Do not update freeID
void remove_com(int c, EdgeCommunities *com, SuperPartition *spart, const graph *g) {
    int sset = spart->com_to_sset[c];

    if (com->nb_edge[c] != spart->nb_edge[sset]) {
     spart->p++;

     spart->com_to_sset[c] = spart->freeID;
     spart->nb_edge[sset] -= com->nb_edge[c];
     spart->nb_node[sset] -= node_diff(c, sset, com, spart, g);

     spart->nb_edge[spart->com_to_sset[c]] = com->nb_edge[c];
     spart->nb_node[spart->com_to_sset[c]] = com->nb_node[c];

       if (spart->nb_node[sset] <= 0 || spart->nb_edge[sset] <= 0)
          report_error("a sset ended up with a non-positive number of nodes or edges after moving a community out of it");
    }
}

// move c from its current super set (local variable ori_sset) to dest_sset
void move(int c, int dest_sset, EdgeCommunities *com, SuperPartition *spart, const graph *g) {
     int diff_nodes_ori;
    int diff_nodes_dest = node_diff(c, dest_sset, com, spart, g);
    int ori_sset = spart->com_to_sset[c];

     spart->nb_edge[dest_sset] += com->nb_edge[c];
    spart->nb_node[dest_sset] += diff_nodes_dest;
```

```

   spart->com_to_sset[c] = dest_sset;


    if (spart->nb_edge[ori_sset] == com->nb_edge[c]) {
        spart->nb_edge[ori_sset] = -1;
        spart->nb_node[ori_sset] = -1;
        spart->p -= 1;
      if (ori_sset < spart->freeID) spart->freeID = ori_sset;
    }
    else {
        diff_nodes_ori = node_diff(c, ori_sset, com, spart, g);
        spart->nb_edge[ori_sset] -= com->nb_edge[c];
        spart->nb_node[ori_sset] -= diff_nodes_ori;
    }

}

// take a partition into edge communities and group some communities together to obtain a superpartition
// return true if there is an improvement
bool one_level_edge(const graph *g, EdgeCommunities* com, SuperPartition *spart, bool* calculated, long double* expectation_table) {
   int c;
   ////////// initialize couples and expectation ////////////////
   // remember to update this for every insert/remove:
   int couples_in_spart = 0;
   long double expect_spart = 0.0;
   int diff_nodes;
   int nodes_sset;

   // all communities in different super sets
   for (c = 0; c < com->k; c++) {
      couples_in_spart += (com->nb_node[c] * (com->nb_node[c] - 1)) / 2;
      expect_spart += expectation(com->nb_edge[c], g, calculated, expectation_table);
   }

   bool improved_this_turn;
   bool overall_improvement = false;
   long double best_mod_gain;
   long double gain_comeback;
   int inter_sset;
   int *random_order;

   int round = 0;
   int h, c2;
   do {
      round++;

      if (RANDOM) {
         random_order = rand_perm(com->k);
      } else {
         random_order = (int*) malloc(com->k*sizeof(int));
         for (int i = 0; i < com->k; i++) {
            random_order[i] = i;
         }
      }
      improved_this_turn = false;
      ////////////// Treat community c /////////////////
      for (h = 0; h < com->k; h++) {
         int c = random_order[h];
         int old_sset = spart->com_to_sset[c];
```

```
/////////// Put c in a sset by itself /////////////////
remove_com(c, com, spart, g);
inter_sset = spart->com_to_sset[c];

// update if we moved c:
if (spart->com_to_sset[c] != old_sset) {
   // update expectation:
   expect_spart -= expectation(com->nb_edge[c] + spart->nb_edge[old_sset], g, calculated, expectation_table);
   expect_spart += expectation(spart->nb_edge[old_sset], g, calculated, expectation_table);
   expect_spart += expectation(com->nb_edge[c], g, calculated, expectation_table);

   // update couples:
   diff_nodes = node_diff(c, old_sset, com, spart, g);
   nodes_sset = spart->nb_node[old_sset];
   couples_in_spart -= ((nodes_sset + diff_nodes) * (nodes_sset + diff_nodes - 1)) / 2;
   couples_in_spart += (nodes_sset * (nodes_sset - 1)) / 2;
   couples_in_spart += (com->nb_node[c] * (com->nb_node[c] - 1)) / 2;
}

/////////////// which sset do we insert c into? //////////////////
int best_sset = -1;
best_mod_gain = -LDBL_MAX;
if (spart->com_to_sset[c] == old_sset) gain_comeback = 0.0;
for (c2 = 0; c2 < com->k; c2++) {
   int new_sset;
   if (c2 != c) {
       new_sset = spart->com_to_sset[c2];
      if (ONLY_NEIGHBOURS && node_diff(c, new_sset, com, spart, g) == com->nb_node[c]){
         continue; //don't move if no links are shared between V(c) and V(new_sset)
      }
    long double gain = 0.0;
    gain = edge_mod_gain(c, new_sset, couples_in_spart, expect_spart, com, spart, g, calculated, expectation_table);
    if (new_sset == old_sset) gain_comeback = gain;

    if (gain > best_mod_gain) {
       best_mod_gain = gain;
       best_sset = new_sset;
    }
 }
}

if (best_mod_gain < 0.0) {
   best_mod_gain = 0.0;
   best_sset = spart->com_to_sset[c];
}

if (best_mod_gain > gain_comeback) {
   improved_this_turn = true;
   overall_improvement = true;
}
   else {
```

```
                best_sset = old_sset;
            }

            //////////////////// insert c into best community or keep intermediate ///////////////
            if (best_sset == inter_sset) {
                if (inter_sset == spart->freeID) {
                    while (spart->nb_node[spart->freeID] != -1) spart->freeID++;
                }
                else {
                }
            } else {
                // insert into new community

                // update expectation
                expect_spart -= expectation(com->nb_edge[c], g, calculated, expectation_table);
                expect_spart -= expectation(spart->nb_edge[best_sset], g, calculated, expectation_table);
                expect_spart += expectation(spart->nb_edge[best_sset] + com->nb_edge[c], g, calculated, expectation_table);

                // update couples
                diff_nodes = node_diff(c, best_sset, com, spart, g);
                nodes_sset = spart->nb_node[best_sset];
                couples_in_spart -= (spart->nb_node[best_sset] * (spart->nb_node[best_sset] - 1)) / 2;
                couples_in_spart -= (com->nb_node[c] * (com->nb_node[c] - 1)) / 2;
                couples_in_spart += ((nodes_sset + diff_nodes) * (nodes_sset + diff_nodes - 1)) / 2;

                // insert c into the best sset:
                move(c, best_sset, com, spart, g);


                // sset with freeID is now free again
                if ((spart->nb_edge[spart->freeID] != 0 && spart->nb_edge[spart->freeID] != -1)
                    || (spart->nb_node[spart->freeID] != 0 && spart->nb_node[spart->freeID] != 0))
                spart->nb_node[spart->freeID] = -1;
                spart->nb_edge[spart->freeID] = -1;
            }
        }
        free(random_order);
    } while (improved_this_turn);
     return overall_improvement;
}

// write output of algorithm to file out.
void output(FILE *out, EdgeCommunities *com, const graph *g, bool *calculated, long double *expectation_table) {
    int couples = count_couples(com);
    long double expected = calculate_expectation(g, com, calculated, expectation_table);

    if (MEASURE == 1) fprintf(out, "Modularity used: m/r - m/E(r)\n");
    else if (MEASURE == 2) fprintf(out, "Modularity used: (E(r) - r)\n");
    else if (MEASURE == 3) fprintf(out, "Modularity used: (E(r)/r)\n");
    else report_error("measure must be given with -m option");
```

```c
   fprintf(out, "Modularity m/r - m/E(r): %Lf\n", edge_modularity(g, com, calculated, expectation_table));
   fprintf(out, "Modularity (E(r) - r): %Lf\n", edge_mod_minus(g, com, calculated, expectation_table));
   fprintf(out, "Modularity (E(r)/r) = %Lf\n", edge_mod_ratio(g, com, calculated, expectation_table));
   fprintf(out, "couples: %d\n",  couples);
   fprintf(out, "expectation: %Lf\n", expected);
   fprintf(out, "# of communities: %d\n", com->k);
   print_communities(g, com, out);
}

// LOUVAIN FOR EDGES, MAIN FUNCTION
EdgeCommunities* edge_louvain(const graph *g, bool* calculated, long double * expectation_table) {
   clock_t start_at = clock();
   int i;
    bool improved = true;
    int* visited_nodes;
    EdgeCommunities* com;
   SuperPartition* spart;

    if( (visited_nodes=(int*)malloc(g->n*sizeof(int))) == NULL )
       report_error("main: malloc() error");
    for (i=0; i<g->n; i++) visited_nodes[i]=-1;

    if( (com=(EdgeCommunities *)malloc(sizeof(EdgeCommunities))) == NULL )
       report_error("main: malloc() error");

   // initialize partition
   init_edge_communities(g, com);
   spart = init_superpart(com);

   // prepare output files
   FILE *out = NULL;
   if ( (out=fopen(OUTPUT_FILENAME,"w"))==NULL)
      perror("fopen");

   FILE *history = NULL;
   if (HISTORY_FILENAME) {
      if ( (history=fopen(HISTORY_FILENAME,"w"))==NULL)
         perror("fopen");
   }

   i=0;
   while (improved) {
    i++;
    improved = one_level_edge(g, com, spart, calculated, expectation_table);
    update_communities(com, spart, g, visited_nodes);
      free_SuperPartition(spart);
    spart = init_superpart(com);

      if (HISTORY_FILENAME) {
         fprintf(history, "after STAGE %d\n", i);
         output(history, com, g, calculated, expectation_table);
      }
   }

   double elapsed = ((double) (clock() - start_at)) / CLOCKS_PER_SEC;
   int elapsed_hour = elapsed / (60*60);
   int rest_min = elapsed/60 - elapsed_hour*60;
   int rest_sec = elapsed - elapsed_hour*60*60 - rest_min*60;
```

```
    fprintf(out, "elapsed time: %d hour, %d min, %d sec, after final stage
        ↪ (%d):\n", elapsed_hour, rest_min, rest_sec, i);
    output(out, com, g, calculated, expectation_table);

    fclose(out);
    if (HISTORY_FILENAME) fclose(history);

    free(visited_nodes);
    free_SuperPartition(spart);

    return com;
}

///////////////////////////
////       MAIN        ////
///////////////////////////

int main(int argc, char **argv) {
    int i;
    // command line arguments
    read_command_line_args(argc, argv);

    if (RANDOM) {
        srand(time(NULL));
    } else srand((unsigned) 102458);

     FILE* infile=NULL;
     graph* g=NULL;

    // Create graph
    if ( (infile=fopen(IN_NAME,"r"))==NULL)
        report_error("IN_NAME -- fopen: error");

    g = graph_from_file(infile);
    fclose(infile);

     long double *expectation_table;
     bool *calculated;

     if( (expectation_table=(long double *)malloc((g->m+1)*sizeof(long
         ↪ double))) == NULL )
         report_error("main: malloc() error");
     for (i=0; i<g->m+1; i++) expectation_table[i] = -1.0;

     if( (calculated=(bool *)malloc(g->m+1*sizeof(bool))) == NULL )
         report_error("main: malloc() error");
     for (i=0; i<g->m; i++) calculated[i] = false;

     EdgeCommunities* com;
     com=edge_louvain(g, calculated, expectation_table);

    free_EdgeCommunities(com);
    return 0;
}
```