NTNU
Det skapende universitet

Community Detection in Large Social Networks

Mats Julian Olsen

Master i fysikk og matematikk
Innlevert: juni 2014
Hovedveileder: Brynjulf Owren, MATH
Medveileder: Kenth Engø-Monsen, Telenor Research

Norges teknisk-naturvitenskapelige universitet
Institutt for matematiske fag

Sammendrag

I denne masteroppgaven implementeres og testes to algoritmer for å finne gruppestruktur i nettverk, nemlig *Louvain*-metoden og *Diffusion and Propagation*-metoden. Et nettverks gruppestruktur består av en naturlig inndeling av nettverkets noder i ikke-overlappende sett, der hvert sett består av noder som er tettere koblet til hverandre, enn til resten av nettverket. De ovennevnte algoritmene er to alternativer blant mange gode iterative teknikker som har sett dagens lys i løpet av de siste 15 årene [31, 32, 5].

Vi presenterer tre endringer til metodene nevnt over. Først introduserer vi en tredje fase i Louvain-metoden, og endrer dens aggregerende natur ved å la metoden bryte opp grupper i tillegg til å slå dem sammen. Videre undersøker vi hva som skjer med den beregnede gruppestrukturen når matrisen som representerer det underliggende nettverket gjennomgår en av flere matrisetransformasjoner. Spesifikt er vi interesserte i se på transformasjoner der kantmatrisen til nettverket opphøyes i andre og tredje potens, samt matriseeksponentialet. Til slutt tester vi og sammenligner resultatene til metodene på genererte nettverk av ulike slag [21, 20, 22], samt to store sosiale nettverk med millioner av noder fra den virkelige verden.

Abstract

The purpose of this master's thesis is first and foremost to implement and benchmark two well-known methods for community detection, namely the Louvain method [5] and the Diffusion and Propagation Algorithm [37]. With a community partitioning we wish to uncover the network's intrinsic subdivision into groups of vertices that are more densely connected with each other, than to the rest of the network. The two above mentioned algorithms are two of many good iterative methods that have been researched over the past 15 years [31, 32, 5].

In this thesis we present three novel alterations to the basic methods. First, we introduce a third phase to the Louvain method, which lets it divide up communities in which vertices are stuck in "local optimas", modifying its aggregating nature slightly. Second, we also present, implement and analyze a naive method of constructing communities around high-degree vertices. Third, we compare the community structure outputted by these methods when the adjacency matrix have been transformed by a different matrix functions, specifically, the matrix exponential and matrix powers. We dodge the added computational cost incurred by the denseness of the outputted matrices, by introducing what we call *edge restriction*. Finally, we benchmark the methods on both weighted and unweighted state-of-the-art computer generated benchmark graphs [21, 20, 22] and on two large real-world social networks.

Acknowledgements

I would like to thank my super supervisor Kenth Engø-Monsen for giving me the opportunity to work on something I am truly passionate about, and providing excellent guidance at that. My family, for believing in me, and always telling me to hang in there. My girlfriend Ine, for her support and for being the best of distractions. Last, but not least, I would like to thank the friends I've made the past five years, Bjørn, Edvard, Kine, Hager, Henrik, Hallvard, Lars, Petter, Gullik and Tor, as well as Andreas and Leon, for making it truly worthwile.

Mats Julian Olsen, Trondheim, June 2014

Contents

1 Introduction

Networks are structures of great importance. As modern human beings, we are surrounded by them every minute of every day. The people we talk to, the bus routes in our city, the Internet and our cell phones are all examples of things that constitute networks. Formally, a network is a set of items together with a set of ties between said items. The ties represent a connection, or interaction, between a pair of items in the network. There is a formidable number of structures that can be considered, or modeled as, a network, and analyzing such structures have never been more popular.

In this thesis we consider a mesoscopic analysis method of networks, known as *community detection*. Along with other methods, community detection compose the increasingly popular field of network analysis. The recent leap in available network data provided by the gains in bandwidth, computational power and storage capacity, has made this field more relevant than ever. We understand by mesoscopic analysis, the analysis of "intermediate" levels of the network. In this age of "big data", where the size of the available networks increase to millions or even billions of actors, analyzing them on a micro level becomes a daunting task, and in fact macro level analysis may be of little value as properties often vary throughout the network. With this increase in network size, assessing the intrinsic, intermediate structures of the network grows more and more important.

Community detection aims to reveal groups of items in our network, that are more closely knit towards each other, than towards the rest of the network. Although communities may arise in all types of networks, our intuitive understanding of them is particularly clear in a *social* network. In a social network, the connected items are social actors, such as persons or organizations, and the ties refer to social interaction or communication between pairs of actors. A group of friends together with their phone records, or internet chat logs, can most certainly be considered a social network. At the same time, it is clear that the group must also be part of a much larger network, and within this larger network they likely constitute exactly what we are looking to find, that is a *community*. Communities may also be defined in a hierarchical matter. The network of all high-school students in some city, for example, may be broken down into communities comprised of students that all go to the same school. These communities may again be refined into groups of friends, or even students that share the same classes. Many methods for finding communities, in fact outputs such a hierarchic structure, and it is often so that each level in the hierarchy has a different interpretation.

For social networking companies such as Twitter or Facebook, the community structure in combination with meta-data may provide valuable insight into how the network is organized. For telecommunication companies, knowing how the customer base clusters together may open up several possibilities. It may provide more efficient ways for communicating with the different customer segments, a way of giving offers to customers that behave and respond in the same manner, or assessing the overall stability of a customer base. The old-school way of commercial clustering based purely on a customer's attributes such as age, gender and address, is clearly challenged when faced with modern community detection that produces information

on how customers actually link together.

In this master's thesis we investigate the state of the art of community detection, implementing both the Louvain method and the Diffusion and Propagation Algorithm. Each one representing the finest of the modularity- and label propagation based approaches, respectively. We continue by presenting ideas for extending the Louvain method, and introduce the stand-alone algorithm Degree-rank. Modern benchmark graphs [22] for testing and benchmarking community detection algorithms are generated, and the algorithms' performances are compared on these networks. We investigate the impact of transforming the adjacency matrix representing the graph in various ways. Finally, we run our algorithms on two real-world datasets provided by Telenor Research, representing all the phone calls and SMSs for millions of Telenor customers during three months.

Small parts of this thesis have earlier been presented in a project entitled "Community Detection in Large Social Networks" [33], where the focus was entirely on the Louvain method. Specifically, the implementation of the Louvain method and parts of Sections 2 and 5, was part of the project report. Building on the project report, much of the notation has been revised, and the implementation of the Louvain method is part of a larger community-detection framework.

This thesis is structured as follows. Section 2 introduces the necessary graph theory, as well as presenting measures borrowed from information theory for comparing community structures. Section 3 introduces, on a theoretical level, the above mentioned community detection algorithms, among them the novel approaches Community Dissolve and Degree-rank. Section 4 proceeds to discuss the implementation of these algorithms, while Section 5 discusses how to generate benchmark graphs and explains necessary pre-processing steps for the real-world data sets. In this section, the matrix transformations aimed to benefit community detection are also presented. The results of the analysis are presented in Section 6, while some concluding remarks are given in Section 7.

2 Theory

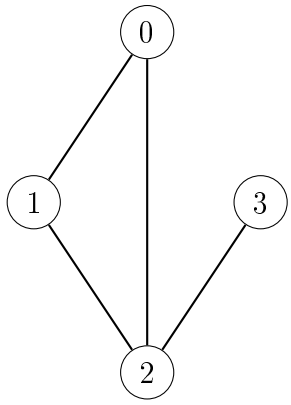
2.1 Social Networks

Conceptually, we understand by the term social network, a structure that maps the interaction between individuals. A social network is a network where the items are social actors, and the ties of the network represent interaction or communication. Examples of social actors include persons, animals, organizations and so on.

The reader is probably aware of several examples of such structures, among them the social networking sites billions of people use daily. It is not so, however, that social networks are an artifact of the internet. In fact there are countless examples of social networks in society, and the recent boom in online social networking sites are merely a natural result of recent gains in computational power and network bandwidth.

Consider, as an example, the flow of letters between households in a city. It is clear that a letter between address A and address B indicates an interaction between the different households, and hence all the addresses together with the letters form

Figure 1: Simple, undirected graph consisting of four vertices and four edges.



a network. If we rather consider the people that sent and received the letters as the interacting items, the resulting structure may be considered a social network.

A class of social network important for this thesis, and which has been studied in great detail [5, 35, 30, 6], is cell phone networks. Here, the actors are phone subscribers, and the ties represent e.g. phone calls or SMSs between two individuals.

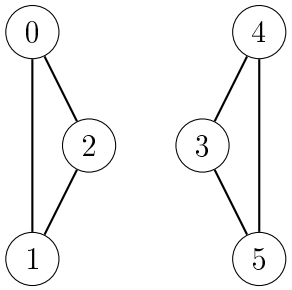
2.2 Graphs

In mathematics, a network is often called a *graph*. A graph, as we know, is a set of objects V together with a set of ties between them, E. We call the connected objects *vertices*, and the set consisting of them, V, the vertex set. The ties are commonly referred to as *edges*, and indicate the connectivity or interaction between two vertices. Formally, an edge is a two element subset of V, and the edge between vertices i and j may be written (i, j). The set holding all edges, E, is called the edge set. An edge may be undirected or directed, indicating a symmetric or asymmetric relationship between two vertices. A graph that has undirected edges and no edges from a vertex to the vertex itself, is called *simple*. An example of a simple graph is shown in Figure 1.

A *weighted* graph G is an ordered triple of V, E and ω, $G = (V, E, \omega)$, where E consists of two-element subsets of V, and ω is a function from $E \to \mathbb{R}^+$ specifying the weight of an edge as a real, positive number. An *unweighted* graphs, is a graph where the function ω is simply the identity, setting the weight of all edges to 1. In this case ω is often omitted in the triple, and we express G as the ordered tuple $G = (V, E)$, instead.

It is clear that an edge represents a relation between vertices, and we usually call this relation the *adjacency* relation, interpreting vertices u and v to be adjacent if and only if $(u, v) \in E$. This leads us to consider the adjacency matrix, A, where we assign integer labels to the vertices such that the ij'th entry marks the weight of the edge between vertex i and j. Usually we denote by n the number of vertices in the graph, and it follows that n is also the number of rows and columns of A. If there is no edge between i and j, the ij'th entry is zero, and if the graph is unweighted, each nonzero entry equals 1. The adjacency matrix of the unweighted, undirected graph in Figure 1 is simply

Figure 2: Disconnected graph with two connected components.



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Note that for an undirected graph the adjacency matrix is symmetric, and that for a simple graph the diagonal elements are always zero. This is because in a simple graph, edges from vertex u to vertex u, i.e. self-loops or simply loops, are not allowed. The diagonal entries of the matrix encodes exactly these loops. Throughout the rest of this thesis, we will not specify ω even if the graph is weighted, as this information is naturally encoded in the adjacency matrix.

The out-degree of a vertex i, is defined to be the sum of the weights of all the edges leaving i. Similarly, the in-degree is the sum of the weights of edges ending at i. If the graph is undirected, the in-degree coincides with the out-degree, and is simply denoted degree. The degree of vertex i will throughout this thesis be denoted as k_i. Given the adjacency matrix of the graph, the out-degree of vertex i is trivially computed as the sum of row i, and correspondingly the in-degree is the sum of column i. Onwards we denote by m half of the sum of all entries in the adjacency matrix, $m = \frac{1}{2}\sum_{ij} A_{ij}$, which for an unweighted graph translates into the number of edges, while for a weighted graph is the sum of the weights, each edge counted exactly once[1].

A walk from vertex u to vertex v of length k is a sequence of vertices $(u_o, \ldots, u_k)$, such that $u_o = u$ and $u_k = v$, and such that each vertex in the sequence is adjacent to both the previous and the next vertex; $(u_i, u_{i-1}) \in E$. If all vertices (and all edges) in the walk are distinct, we say that the walk is a (simple) path.

A graph may or may not be *connected*. If the graph is connected, there exists a path from any vertex i, to all other vertices j in the graph. If this is not true, we say that the graph is disconnected. A connected graph has one *connected component*, the graph itself. A connected component is defined as a subset of the vertices $V_C \subseteq V$, for which any two vertices are reachable with a path, and such that no vertex in V_C is connected to any vertex in $V \setminus V_C$. As such, a disconnected graph may have several connected components, each of them defined as above. Figure 2 shows a disconnected graph with two connected components.

[1] In fact, we are really count each edge twice, but then we divide by 2 to account for it.

2.3 Community Detection

Any network may be analyzed on micro-, meso- and macroscopic levels, all of which provides different, but more or less meaningful information about the network. At the microscopic level one might investigate the properties of the edges between any two actors; the strength of, reciprocity of and the number of such ties in the network. At the macro level, one might be interested in the degree distribution or the diameter[2] of the network. However, when the networks in question grow large, the information at a meso level becomes increasingly important. At the mesoscopic level, we consider how the network is structured; for example how vertices group together into dense clusters known as *communities*. The exercise which is finding these communities is known as community detection, and it is an increasingly popular way of analyzing networks at the mesoscopic level.

Abstractly, community detection is the task of dividing the vertex set of a network into subsets such that the connections within these subsets are denser than the connections between them. These subsets are called communities, or clusters, and each community is a subset of vertices that are more tightly knit together with each other, than to the rest of the graph. We define a non-overlapping *community structure*, or *clustering*[3], as set of communities such that all vertices are included in exactly one community. In other words, a community structure, C, on the graph $G = (V, E)$ is defined as

$$C = \{c_1, c_2, \ldots c_{n_C}\} \text{ s.t. } c_i \cap c_j = \emptyset \text{ and } \cup_{i=1}^{n_C} c_i = V, \tag{1}$$

where c_i is a non-empty subset of V and n_C denotes the number of communities in C. Throughout this thesis we'll assume that a community detection algorithm is a procedure that accepts as input a graph G or its adjacency matrix A, and outputs a community structure C. Some methods for community detection outputs a hierarchical community structure. We define here a hierarchical community structure to be a sequence of community structures $(C_1, C_2, \ldots)$ such that C_{i+1} can be obtained by merging communities in C_i. The condition is equivalent with C_i having to be a *refinement* of C_{i+1}, that is, that C_i is obtained by splitting some of the communities of C_{i+1} [29]. As a result, $n_{C_{i+1}} < n_{C_i}$.

Depending on the initial graph, communities might indicate anything from groups of friends within a social networking site, to a family in the call data of a cell phone provider or even books on amazon treating the same topic [9]. In some cases the structure found may be evidence of groups of which we had no prior knowledge, providing new information about the structure of the network.

2.4 Modularity

The modularity, Q, of a network divided into communities, is the sum of the fraction of edges within communities minus the expected fraction of such edges if the edges

[2] The length of the longest shortest path between any two vertices.

[3] We will use the terms *community structure* and *clustering* interchangeably throughout this thesis, interpreting a clustering as a *vertex* clustering

were distributed at random. Modularity successfully encodes the underlying intuitive feature of a community; that vertices within one are more densely connected to each other, than to the rest of the network. The modularity of a graph $G = (V, E)$ is expressed as

$$Q = \sum_{ij} \left[\frac{A_{ij}}{2m} - e_{ij} \right] \delta_{c_i c_j}, \tag{2}$$

where A_{ij} denotes the ij'th entry of the adjacency matrix A of the graph, $m = \frac{1}{2}\sum_{ij} A_{ij}$ the total weight in the graph (or number of edges if the graph is unweighted), $\delta_{c_i c_j}$ is 1 if the community of i, c_i, is equal to the community of j, c_j, and zero otherwise. The expected fraction of edges falling between two vertices i and j is e_{ij}, and is given by a chosen null model.

2.4.1 Null models

A null model of a graph G is a another graph, G', that matches G in some structural feature, but otherwise is an instance of a random graph. In principle, we may choose any null model when calculating the modularity, but certain models have properties that make them better choices than others. Among these desired properties, are the similarity between G' and G, and the ease-of-computation of e_{ij}.

The simplest choice is the standard Bernoulli random graph. In the Bernoulli random graph, edges appear uniformly with probability p between all vertex-pairs, and this satisfies our demand for ease-of-computation. However, the model is not a good representation of a real-world network [32]. This becomes especially apparent when we compare this random graph's binomial (Poisson for large graphs) degree distribution with real-world degree distributions, which for network data often is observed to follow Power Laws [2] or even Lognormals [35].

The configuration model is another choice. It produces a random graph very similar in structure to the original graph, by realizing the identical degree sequence $(k_1, k_2, \ldots, k_n)$ of the graph G, but placing the edges at random. The drawback of this model is that the probability of having an edge between vertex i and j is especially hard to calculate due to the dependency of the edges [7].

To address this issue Chung and Lu proposed a variation of the configuration model in which instead the random graph G''s *expected* degree sequences, rather than its actual degree sequence, matches the given degrees of G [8, 7].

2.4.2 The Chung-Lu Variation

While the configuration model generates a random graph that realizes a provided degree sequence, the Chung-Lu variant generates a random graph with a given *expected* degree sequence. Given an observed graph $G = (V, E)$ with n vertices, the Chung-Lu null model will construct a graph with expected degree sequence identical to the degrees, $(k_1, k_2, \ldots, k_n)$, of the original graph, and where the edges are placed at random.

Assume that the probability that there exists an edge between vertex i and j in the random graph is P_{ij}. The expected degree of vertex i is then given as the sum of

these probabilities for all j. As the model assumes the expected degree to be equal to k_i, we have that

$$\sum_j P_{ij} = k_i. \tag{3}$$

To calculate the probability that a vertex i is connected to a vertex j given the expected degree sequence, we cut all edges at the middle and consider them as stems. First, we note that when edges are placed totally at random, the probability of choosing a stem of i is only dependent on the expected degree k_i, and that the probabilities of two stems connecting to each other are two independent probabilities multiplied together [31]. Hence we may think of P_{ij} as the product of some density function evaluated at k_i and k_j. Equation (3) can now be written as

$$\sum_j P_{ij} = f(k_i) \sum_j f(k_j), \tag{4}$$

such that $f(x) = Cx$ for some constant C. The implicit constraint that the sum of the degrees must equal the number of edges in the graph, m, or equivalently that $\sum_{ij} P_{ij} = 2m$ yields

$$2m = \sum_{ij} P_{ij} = \sum_{ij} C^2 k_i k_j = (2mC)^2. \tag{5}$$

This gives $C = \frac{1}{\sqrt{2m}}$ and hence the expected fraction of edges between vertex i and j is $e_{ij} = \frac{P_{ij}}{2m} = \frac{k_i k_j}{(2m)^2}$. The modularity may be written as

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta_{c_i c_j}. \tag{6}$$

We must also assume that $k_i^2 < 2m$ such that P_{ij} is always less than 1, so that it encodes a valid probability. In practice, this condition may not be met by networks consisting of only a few vertices and edges with high weights, but for larger networks this impose no problem. Technically, we say that the Chung-Lu variation is a random graph conditioned on the expected degree sequence, whereas the configuration model is conditioned on the actual sequence. In the limit of large networks, the probability of the edges of the configuration model approaches that of the Chung Lu variant [31].

As a side note, observe that (6) may also be written as a sum over all the different communities in our community structure C:

$$Q = \frac{1}{2m} \sum_{c \in C} \sum_{ij \in c} \left[A_{ij} - \frac{k_i k_j}{2m} \right]. \tag{7}$$

In fact, we may also write this as

$$Q = \sum_{c \in C} Q_c, \tag{8}$$

letting Q_c be the modularity of community c given by

$$Q_c = \frac{1}{2m} \sum_{i,j \in c} \left[A_{ij} - \frac{k_i k_j}{2m} \right]. \tag{9}$$

2.5 Comparing Community Structures

It's clear that modularity encodes in some way how significant a community structure is. Modularity takes the value 0 when all vertices are placed in a single community, and often negative values when all vertices are in their own community. Higher values of modularity indicates a good community structure, and the value may approach 1 if the network admits a close to perfect community structure. All networks may have a theoretical maximum value of modularity, which in fact may be far from 1, but this maximum value of modularity is an intrinsic property of the network, not the algorithm that outputted the community structure. This means, among other things, that we, if we don't brute force the solution, can never be sure what this maximum value of the network really is.

All community structures C have an associated modularity Q_C. Given a known *ground truth* community structure C^*, can we assess the quality of some outputted community structure C' by means of its modularity $Q_{C'}$? The answer is not clear. Sure, we may compare two values of modularity, but saying that some algorithm outputted a structure that had 0.099 less modularity than the ground truth structure, really has no clear interpretation. It's also perfectly possible for two different community structures to obtain the same modularity. After all, modularity is not a measure of the similarity of clusterings.

When benchmarking community detection algorithms on networks with a priorly known community structure, what we want is some measure of "how far" the outputted structure was from the known, ground truth structure. In other words, we wish to measure the distance between to clusterings. As it turns out, this is not an uncommon problem in information theory, and we shift our focus there for the rest of this section, however always trying to ground the theory in our application to community detection.

2.5.1 Empirical Probability Distributions

Consider the graph $G = (V, E)$ and two community structures C and C' of length n_C and $n_{C'}$, respectively. Let X be a random variable that draws a label at random from the above defined clustering C. The (empirical) probability of X taking label x is the relative frequency of x in C

$$p(X = x) = p(x) = \frac{n_x}{n}, \tag{10}$$

where n_x denotes the number of vertices labeled x in the community structure C and $n = \mid V \mid$. The same holds for a random variable Y drawing labels from C'. The $n_{C'} \times n_C$ confusion matrix N defined by

$$N = \begin{bmatrix} N_{1,1} & \cdots & N_{1,n_C} \\ \vdots & \ddots & \vdots \\ N_{n_{C'},1} & \cdots & N_{n_{C'},n_C} \end{bmatrix} \tag{11}$$

where element N_{ij} indicates the number of vertices that lies in community i in C' and j in C, is known as the contingency table of X and Y. By dividing every element in N by n, the total number of vertices in the graph, we have the joint distribution of the variables drawing labels from C and C'. Hence we have just defined the probability mass function

$$p(c, c') = \frac{1}{n} N_{c,c'}, \tag{12}$$

and it tells us the probability that a vertex in our network is given label c in C and c' in C'. The sums $p(c) = \sum_{j \in C'} p(c, j)$ for $c \in C$ defines the marginal distribution of X, the random variable that draws its label from C, as seen above. Doing the corresponding sum over C gives the corresponding marginal distribution for Y. We know that $p(c)$ and $p(c, c')$ are valid probability distributions since $0 < N_{ij} < n$ and $\sum_j N_{ij} = \sum_i N_{ij} = 1$ for all j and i respectively.

2.5.2 Entropy

We've now seen how we can interpret the output of a community detection algorithm by means of probabilities and densities. Let us now define an important concept in information theory that measures the uncertainty in the outcome of a random variable, namely the entropy. The entropy of a random variable X taking values x in C is defined as

$$H(X) = -\sum_{x \in C} p(x) \log p(x). \tag{13}$$

The entropy of a clustering C takes the value 0 if and only if the clustering has only one partition. This we interpret as C having no uncertainty in what community some vertex u belongs to. We may also consider the entropy of X conditioned on the outcome of another random variable Y by using conditional probabilities:

$$\begin{aligned} H(X \mid Y) &= \sum_{y \in C'} p(Y = y)\, H(X \mid Y = y) \\ &= -\sum_{y \in C'} p(Y = y) \sum_{x \in C} p(X = x \mid Y = y) \log p(X = x \mid Y = y). \end{aligned}$$

From this we may write the joint entropy,

$$H(X, Y) = -\sum_{x \in C, y \in C'} p(x, y) \log p(x, y), \tag{14}$$

where $p(x, y)$ is shorthand for $p(X = x, Y = y)$, as

$$
\begin{aligned}
H(X,Y) &= -\sum_{\substack{x\in C\\ y\in C'}} p(x,y)\log[p(x\mid y)p(y)] \\
&= -\sum_{\substack{x\in C\\ y\in C'}} p(x,y)\left(\log p(x\mid y)+\log p(y)\right) \\
&= -\sum_{\substack{x\in C\\ y\in C'}} p(x,y)\log p(x\mid y) - \sum_{\substack{x\in C\\ y\in C'}} p(x,y)\log p(y) \qquad (15)\\
&= -\sum_{y\in C'}[p(y)\sum_{x\in C} p(x\mid y)\log(p(x\mid y))] - \sum_{y\in C'} p(y)\log p(y) \qquad (16)\\
&= H(X\mid Y)+H(Y).
\end{aligned}
$$

From (15) to (16) we use the definition of marginal probability. The above is known as the "chain rule" of entropy. Having defined conditional and joint entropy, we're now ready to look at some ways to judging the distance between two community structures.

2.5.3 Mutual Information

Mutual information is a measure of two variable's mutual dependence. Formally, we define the mutual information of random variables X and Y taking values from C and C' as

$$I(X,Y) = \sum_{x\in C}\sum_{y\in C'} p(x,y)\log\frac{p(x,y)}{p(x)p(y)}. \qquad (17)$$

If we're given a random vertex $u \in V$, and we're interested in the uncertainty in its community in C, it is measured by the entropy of X, $H(X)$, as X is the random variable taking values from C. Now, if we're told what community u belongs to in C', does the uncertainty in what community u belongs to in C change? It might, and this change is measured by the mutual information of X and Y, which is the specified change averaged over all pairs of communities in C and C' [29]. We may represent different measures of the information of two variables X and Y in a *information diagram*, see Figure 3. The illustration makes among others the following identity clear:

$$I(X,Y) = H(X)+H(Y)-H(X,Y). \qquad (18)$$

We should keep in mind that distances like the mutual information are rarely used one by one. Given a graph G and a ground truth community structure, C, we wish to compare the outputs of algorithms A and B, C_A and C_B pairwise against C using some measure. If the algorithms are not deterministic, often the results of the pairwise comparison are averaged over several runs. Hence the distances between the community structures are subject to addition and subtraction [29]. Adding distances between clusterings measured in mutual information does not have a clear

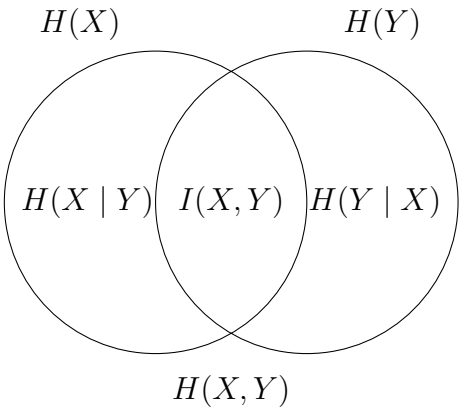


Figure 3: Basic quantities in information theory

interpretation. This is because mutual information is not a metric[4]; it does not obey the triangle inequality [10].

2.5.4 Variation of Information

Variation of information is another measure of the distance between two clusterings. It's a true metric, and a simple linear expression involving entropies and the mutual information of X and Y:

$$VI(X,Y) = H(X) + H(Y) - 2I(X,Y) = H(X,Y) - I(X,Y). \tag{19}$$

We may think of $VI(X,Y)$ as how much knowing the outcome of X reduces the uncertainty of Y. In other words, if the variation of information is high, the partitionings C and C' are very different. If it's approaching zero, they are almost the same. The variation of information is a true metric [29, 28] on clusterings. The fact that the triangle inequality holds, tells us that if two community structures are close to a third, they have to be close to each other. By looking at figure 4 we see the useful identity

$$VI(X,Y) = H(X \mid Y) + H(Y \mid X) \tag{20}$$

For a thorough introduction to variation of information the reader is directed to [29].

2.5.5 Normalization

Both the mutual information and the variation of information are measures whose outcome depend on the number of possible outcomes of the involved random variables [28, 29, 26]. As such, comparing community structures of different sizes does not have a clear meaning. To be able to do such a comparison, we normalize the measures making them output values between 0 and 1. If D is some measure of distance, normalization is typically done by the division of some constant c

[4]The reader is referred to `http://en.wikipedia.org/wiki/Metric_(mathematics)` for a quick introduction to metrics.

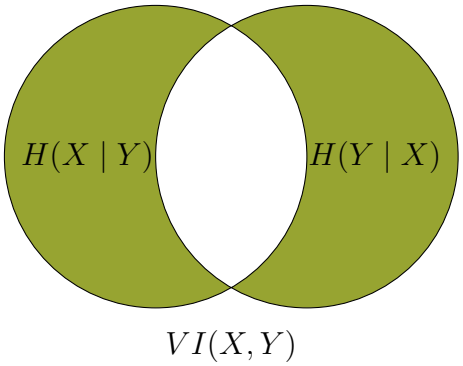


Figure 4: Variation of information is colored green.

$$D_{\text{norm}} = \frac{D}{c},$$

such that c is always great than D, leaving D_{norm} a measure within the unit interval. We proceed to review a few proposed normalizations to both the mutual information and the variation of information.

Sum-normalization of mutual information
Mutual information defined in (18) may be normalized by $\frac{2}{H(X)+H(Y)}$

$$I(X,Y)_{sum} = 2 \cdot \frac{I(X,Y)}{H(X)+H(Y)}. \tag{21}$$

as proposed in [11].

Max-normalization of mutual information
We may also normalize (18) by the maximum of $H(X)$ and $H(Y)$[26]

$$I(X,Y)_{max} = \frac{I(X,Y)}{\max(H(X),H(Y))}. \tag{22}$$

It was proposed as a as a fix to (21), as (21) may overestimate the similarity between clusters.

Normalization of variation of information
The variation of information from (19) may be normalized as

$$VI(X,Y)_{norm} = \frac{1}{2}(\frac{H(X \mid Y)}{H(X)} + \frac{H(Y \mid X)}{H(Y)}), \tag{23}$$

as done in [11]. It can be interpreted as the average lack of inferring X given Y.

Joint entropy-normalization of variation of information
We may also normalizes (19) as T

$$VI(X,Y)_{joint} = \frac{VI(X,Y)}{H(X,Y)} = 1 - \frac{I(X,Y)}{H(X,Y)}, \tag{24}$$

as in [38]. In Section 6 we'll use this normalization, when the results are marked normalized variation of information.

3 Methods for Community Detection

3.1 Louvain Method

The Louvain method method, proposed by Blondel et. al. [5], is an efficient algorithm that finds a heuristic structure of communities. It consists of two phases which are repeated iteratively until the community structure has been sufficiently revealed. When the two phases have been completed we say that the algorithm has completed a *pass* and the algorithm does at leas two such passes successively.

3.1.1 First Phase

The first phase begins by placing all vertices in their own community. We then loop through all vertices and consider all the neighbors of vertex i, that is, all the vertices j such that A_{ij} is nonzero, and calculate the gain of removing i from its community and placing it in the community of j, c_j. The vertex i is then put in the community c_j for which the increase in modularity is largest. If none of the potential reassignments of i into other communities are associated with positive gains in modularity, i stays in its original community and the algorithm moves on to the next vertex. The loop is repeated until no further improvements are obtained, i.e. when the modularity has reached a local optima. In practice, having a stopping criterion (threshold) based on the absolute change of modularity during one full loop can boost the speed of the algorithm, with an accompanying loss of quality in the partitioning.

3.1.2 Second Phase

In the second phase of the algorithm, a new network is constructed with the communities from the first phase as vertices. The weights of the edges between the new vertices are given by the sum of the weights between all vertices in the two old communities. The edges and vertices within a community lead to loops in the new network, weighted by the total edge weight between the included vertices. When this procedure is finished, the algorithm has completed what we previously called a pass, and it jumps to first phase in order do several more passes to create a hierarchy of communities. The algorithm stops when a maximum of the modularity is obtained, or in practice, when the last performed pass did not increase the modularity.

3.1.3 Calculating Modularity

The Louvain method is in its essence a method that moves vertices sequentially from their present community to their best match. A vertex' best match is simply the community that, with the addition of the vertex considered, increases the modularity of the network the most.

Let us consider the potential reassignment of a vertex i into the community c, forming the community $c^* = \{c, i\}$ We may think of the modularity of C^* as

$$Q_{c^*} = Q_c + \Delta Q_i^c, \tag{25}$$

where we consider ΔQ_i^c to be amount of modularity i adds to the modularity of community c when joining it. To find the modularity of c^* we first introduce some notation. We write $A_c = \sum_{j,k \in c} A_{jk}$ for twice the sum of the weights inside community c, $A_{ic} = \sum_{j \in c} A_{ij}$ the sum of the weights from i to vertices in c and $k_c = \sum_{j \in c} k_j$ the sum of the degrees of vertices in c. Putting this into (7) we get

$$Q_{c^*} = \left[\frac{A_c + 2A_{ic} + A_{ii}}{2m} - \left(\frac{k_c + k_i}{2m} \right)^2 \right]. \tag{26}$$

Expanding the quadratic term and grouping the terms leaves us with

$$Q_{c^*} = \left(\frac{A_c}{2m} - \frac{k_c^2}{(2m)^2} \right) + \left(\frac{2A_{ic}}{2m} - \frac{2k_c k_i}{(2m)^2} \right) + \left(\frac{A_{ii}}{2m} - \frac{k_i^2}{(2m)^2} \right). \tag{27}$$

Now, we introduce $q_{xy} = \left(\frac{2A_{xy}}{2m} - \frac{2k_x k_y}{(2m)^2} \right)$ such that $Q_{\{x,y\}}$ can be expressed as

$$Q_{\{x,y\}} = Q_x + q_{xy} + Q_y, \tag{28}$$

interpreting q_{xy} as the modularity the pair of vertices x and y generates in addition to their individual modularities $Q_x = \frac{A_{xx}}{2m} - (\frac{k_x}{2m})^2$ and Q_y. Note that by definition q_{xx} to be $2Q_x$. With this in mind we may write

$$Q_{c^*} = Q_c + q_{ic} + Q_i. \tag{29}$$

Here c is a set, and we interpret that

$$q_{ic} = \sum_{v \in c} q_{iv}. \tag{30}$$

Hence, from (29) we conclude that ΔQ_i^c from (25) must be defined by

$$\Delta Q_i^c = q_{ic} + Q_i. \tag{31}$$

We may find a similar expression for the loss of modularity when a vertex i is removed from its community d. Now we're looking for the quantity ΔQ_i^d such that

$$Q_{d'} = Q_d - \Delta Q_i^d. \tag{32}$$

Now realize that we may write, following the notation from before,

$$Q_{d'} = \frac{A_d - 2A_{id} + A_{ii}}{2m} - \left(\frac{k_d - k_i}{2m} \right)^2, \tag{33}$$

noting that we're adding $A_{ii}/2m$ since at the time of the calculation, i is included in d, and hence $2A_{id}$ includes $2A_{ii}$, which is indeed twice the amount we should subtract. Equation (33) can be expanded and regrouped into

$$Q_{d'} = \left(\frac{A_d}{2m} - \frac{k_d^2}{(2m)^2} \right) - \left(\frac{2A_{id}}{2m} - \frac{2k_i k_d}{(2m)^2} \right) + \left(\frac{A_{ii}}{2m} - \frac{k_i^2}{(2m)^2} \right). \tag{34}$$

Again, letting $q_{id} = \left(\frac{2A_{id}}{2m} - \frac{2k_i k_d}{(2m)^2}\right)$ be the sum of the pairwise additional modularities between i and the vertices of d we have that

$$\Delta Q_i^d = q_{id} - Q_i. \tag{35}$$

Now, if we want to know how much the modularity of the network as a whole has changed, we may simply add $Q_{d'} - Q_d$ and $Q_{c'} - Q_c$ to obtain

$$\Delta Q = (Q_{c'} - Q_c) + (Q_{d'} - Q_d) = q_{ic} - q_{id} + 2Q_i \tag{36}$$

3.1.4 Creating the new network

Consider the matrix $S \in \{0,1\}^{n \times n_c}$, where n_c denotes the number of communities. Let S_{ij} be equal to 1 if vertex i is in community j, and 0 otherwise. The new network with the old communities as vertices can now be created as

$$A^* = S^T A S. \tag{37}$$

Thus if s_i is the i'th column of S, we have that

$$A^* = \begin{bmatrix} s_1^T \\ \vdots \\ s_{n_c}^T \end{bmatrix} A \begin{bmatrix} s_1, & \ldots, & s_{n_c} \end{bmatrix} = \begin{bmatrix} s_1^T A s_1 & \cdots & s_1^T A s_{n_c} \\ \vdots & \ddots & \vdots \\ s_{n_c}^T A s_1 & \cdots & s_{n_c}^T A s_{n_c} \end{bmatrix}, \tag{38}$$

and A^*_{ij} is the sum of the entries of A from community i into community j. Observe also that A^* is now a $n_c \times n_c$ matrix, which is the number of vertices in the next pass.

3.2 Dissolving Communities

We here present an idea based on "dissolving" communities. Dissolving a community means moving all vertices from the community, to their best possible alternatives. The Louvain method has a very aggregating nature, its second phase can only merge communities. In Section 3.2.4 we'll introduce a phase in between the first and second phase of the Louvain method, that focuses on the diffusion of the vertices within a community to neighboring communities.

3.2.1 Motivation

The idea of dissolving a community requires a shift in focus from sequentially moving vertices into doing so simultaneously. Considering vertices simultaneously loosens the modularity gain criteria we saw for the Louvain method in Section 3.1.3 for each vertex, and in turn may help "stuck" vertices move from their communities. We saw in Equation (9) that modularity is a quantity defined for communities, as well as for the network as a whole. The modularity of individual communities is a local quantity; it only depends on the involved vertices. Consider the scenario presented in Figure 5. Two vertices u and v comprise a community c, with modularity

$$Q_c = Q_u + q_{uv} + Q_v \tag{39}$$

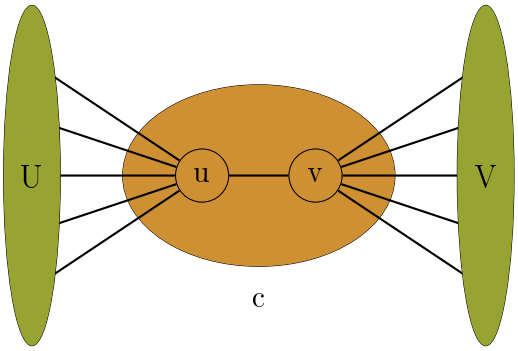


Figure 5: Two vertices u and v constituting the community c.

Now consider the loss of modularity c experiences when we move u, ΔQ_u^c. If we consider moving u as in the Louvain method, we find that if we want to move u from c to some other community U, ΔQ_u^U must be larger than ΔQ_u^c. After all we need the modularity of U to grow more than the modularity of c drops. This translates into the following set of inequalities:

$$\begin{aligned} \Delta Q_u^U &> \Delta Q_u^c \\ q_{uU} + Q_u &> q_{uv} + Q_u \\ q_{uU} &> q_{uv}. \end{aligned} \tag{40}$$

Of course, the above situation is totally analogous for moving v. If we instead consider moving u and v at the same time, we see that the loss in modularity in c cannot exceed Q_c. In other words, for a positive modularity gain, we want that $\Delta Q_u^U + \Delta Q_v^V > Q_c$, since the modularity of c is zero after we've moved the two vertices. Examining closer, this means that

$$\begin{aligned} \Delta Q_u^U + \Delta Q_v^V &> Q_c \\ q_{uU} + Q_u + q_{vV} + Q_v &> Q_u + q_{uv} + Q_v \\ q_{uU} + q_{vV} &> q_{uv}, \end{aligned} \tag{41}$$

and we see how q_{uU} and q_{vV} now added together must be larger than the same quantity they before had to exceed individually.

3.2.2 Local Modularity Changes

Each move in the Louvain method requires the global modularity gain to be calculated, and in Section 3.1 we saw how to calculate it after first finding the local gains. When dissolving communities we're also interested in the global modularity, but then only as a sum over the modularity of the communities. What really matters is the modularity of individual communities before and after moves happen. This shift in mindset from global to local change, is interesting, but we have to take special care as the modularity gains associated with a vertex u is dependent on all other vertices v. Following this paragraph are two paragraphs of modularity-considerations.

Moving two vertices

Moving two vertices u and v at the same time, we have to be careful as ΔQ_u^c and ΔQ_u^d may depend on v and u, respectively. Let's now see how we can reach an expression for the modularity gains of the receiving community and the community the vertices leave. The modularity of a community c with vertices u and v added to it, denoted c^*, can be found analogously as for the single vertex case. The following equation is our starting point.

$$Q_{c^*} = \frac{A_c + 2A_{uc} + 2A_{vc} + 2A_{uv} + A_{uu} + A_{vv}}{2m} - \left(\frac{k_c + k_u + k_v}{2m}\right)^2. \tag{42}$$

By expanding the quadratic term in k and grouping the terms to match known quantities, we have that

$$\begin{aligned} Q_{c^*} = &\left(\frac{A_c}{2m} - \frac{k_c^2}{(2m)^2}\right) + \left(\frac{2A_{uc}}{2m} - \frac{2k_uk_c}{(2m)^2}\right) + \left(\frac{A_{uu}}{2m} - \frac{k_u^2}{(2m)^2}\right) + \\ &\left(\frac{2A_{vc}}{2m} - \frac{2k_vk_c}{(2m)^2}\right) + \left(\frac{A_{vv}}{2m} - \frac{k_v^2}{(2m)^2}\right) + \left(\frac{2A_{uv}}{2m} - \frac{2k_uk_v}{(2m)^2}\right). \end{aligned} \tag{43}$$

Keeping the order of terms in (43), we see that it may be written

$$Q_{c*} = Q_c + q_{uc} + Q_u + q_{vc} + Q_v + q_{uv}, \tag{44}$$

which in turn may be written nicely as

$$Q_{c*} = Q_c + \Delta Q_u^c + \Delta Q_v^c + q_{uv}, \tag{45}$$

or equivalently as

$$Q_{c*} = Q_c + q_{uc} + q_{vc} + Q_{\{u,v\}}. \tag{46}$$

Similarly, for the case of two vertices u and v leaves a community d, we have that

$$Q_{d'} = \frac{A_d - 2A_{ud} - 2A_{vd} + 2A_{uv} + A_{uu} + A_{vv}}{2m} - \left(\frac{k_d - k_u - k_v}{2m}\right)^2. \tag{47}$$

The reason for the plus signs in in the first fraction, is that A_{uv} and A_{uu} are included in A_{ud} as u is a part of d. The same goes for the terms including v. Again, we expand the quadratic term and get

$$\begin{aligned} Q_{d'} = &\left(\frac{A_d}{2m} - \frac{k_d^2}{(2m)^2}\right) - \left(\frac{2A_{ud}}{2m} - \frac{2k_uk_d}{(2m)^2}\right) + \left(\frac{A_{uu}}{2m} - \frac{k_u^2}{(2m)^2}\right) \\ &- \left(\frac{2A_{vd}}{2m} - \frac{2k_vk_d}{(2m)^2}\right) + \left(\frac{A_{vv}}{2m} - \frac{k_v^2}{(2m)^2}\right) + \left(\frac{2A_{uv}}{2m} - \frac{2k_uk_v}{(2m)^2}\right). \end{aligned} \tag{48}$$

Simplifying, and following the same ordering as above, we find

$$Q_{d'} = Q_d - q_{ud} + Q_u - q_{vd} + Q_v + q_{uv}. \tag{49}$$

Again, we may write this as

$$Q_{d'} = Q_d - \Delta Q_u^d - \Delta Q_v^d + q_{uv}, \tag{50}$$

or simply

$$Q_{d'} = Q_d - q_{ud} - q_{vd} + Q_{\{u,v\}}. \tag{51}$$

We might have expected the signs of q_{uv} in (50) and $Q_{\{u,v\}}$ in (51) to be negative. The reason they are not, is that within ΔQ_u^d and ΔQ_u^d lies $2q_{uv}$, and hence we need to add it once to subtract the correct value from Q_d.

Moving a set of vertices

We generalize without the above rigor and have that the modularity of c with the set s added to it is

$$Q_{c*} = Q_c + \sum_{u \in s} q_{uc} + Q_s. \tag{52}$$

However, if we extend the previous definition of q_{xy} when x or y is a set/community to be valid for when both x and y are sets, such that

$$q_{sc} = \sum_{x \in s} \sum_{y \in c} q_{xy}, \tag{53}$$

we have that

$$Q_{c*} = Q_c + q_{sc} + Q_s. \tag{54}$$

The community that loses s, d, has a new modularity, Q'_d, of

$$Q_{d'} = Q_d - q_{sd} + Q_s. \tag{55}$$

Let's check the edge case when $s = d$. Now d' will be empty, and should have zero modularity. Now, according to the above notation, the quantity q_{dd} is given as

$$\begin{aligned} q_{dd} &= \sum_{x \in d} \sum_{y \in d} q_{xy} \\ &= \sum_{x \in d} 2Q_x + \sum_{x,y \in d, x \neq y} q_{xy} \qquad (56) \\ &= 2Q_d. \qquad (57) \end{aligned}$$

From (56) to (57) we realize that within $\sum_{x,y \in d, x \neq y} q_{xy}$ each q_{xy} is counted twice; both as q_{xy} and q_{yx}. Hence we have that (55) simply becomes

$$Q_{d'} = Q_d - 2Q_d + Q_d = 0, \tag{58}$$

when $s = d$.

3.2.3 Moving Criteria

We saw above how to calculate the modularity when moving vertices. Now the trick is to figure out when we wish to dissolve a community. When dissolving a community c, we lose all the modularity of c, Q_c, as illustrated in and between (55) and (58). Letting D be the set of destinations, and $d \in D$ the destination to which some subset of c is moving to. The latter subset we define as $c_d = \{x \in c | x \to d\}$. Then the gain in global modularity is given by

$$\Delta Q = \sum_{d \in D} (q_{c_d d} + Q_{c_d}) - Q_c. \tag{59}$$

Hence, our criteria for dissolving a community becomes

$$\sum_{d \in D} (q_{c_d d} + Q_{c_d}) > Q_c. \tag{60}$$

3.2.4 Extending the Louvain Method

The Louvain method consists of two phases that together comprise what is called a pass. We now introduce a dispersing phase, placed between the first and the second phase of the original Louvain method. This step uses the techniques for dissolving communities defined above, and considers all communities in order of increasing modularity.

Algorithm 1 Community-Dissolve Algorithm

1: Obtain the community, c, with the lowest modularity Q_c which has not yet been considered.
2: For each vertex v in the community, determine the community, d_v of which including v would experience the highest jump in Q_{d_v}.
3: If the sum of the gains for all vertices in c is greater than the modularity of c, Q_c; dissolve the community c.
4: Repeat until all communities have been considered.

Algorithm 1 describes the introduced third phase in the Louvain method. For later reference, we call the Louvain method with the introduced pass, *Community-dissolve*. Note that step 3 in Algorithm 1 may introduce new communities comprised by only a single vertex. These single-vertex community will however be considered almost immediately in step 1, as such a community does not have high modularity.

3.3 Degree-Rank Algorithm

Yet another idea for extending the Louvain method, is including our knowledge of the degree-sequence of the graph. Communities often may be based around high-degree vertices, as high-degree vertices are though to be more *central* in the network [12]. This knowledge has been taken into account in among others the label propagation algorithm we'll see in Section 3.4, where it is sometimes used as giving preference to high-degree vertices. We will here try to use the degree-sequence in a different way in order to establish communities around the vertices with the highest degree. The following method is not an extension of the Louvain method, in the same way as Community-dissolve. The Louvain method considers the neighbors of each vertex, trying to determine which community to include said vertex in. The Degree-rank algorithm, however, considers the neighbors of a high-degree vertex, trying to determine if the modularity of the network will increase by including the neighbor in the community of the vertex.

Algorithm 2 Degree-rank

1: Sort the vertices from high degree to low degree
2: Consider the vertex, u with the highest degree, that has not yet been considered.

3: For each neighbor v of u, calculate the modularity gain from moving v into the community of u. If this gain is positive, move the vertex. Mark v as seen.
4: If there are vertices that are not seen; go to line 2. If not, mark all vertices as not seen and restart on line 2. The algorithm finishes when a the above steps did not increase the modularity more than some provided treshold.

3.4 Label Propagation

So far we've seen community detection algorithms that try to maximize the modularity-function. It should be made clear, though, that the modularity approach is not the only feasible approach for revealing communities. Another branch of methods characterized as label propagation algorithms, has gained ground after an algorithm with close to $\mathcal{O}(m)$ complexity[5] was proposed in [34]. An important thing to note, though, is that label propagation has not been generalized into taking into account weighted graphs. We also would like to specify that throughout this thesis we will use the terms *label* interchangeably with *community*, as they for this application mean exactly the same.

3.4.1 General Label Propagation

The general label propagation algorithm is the basis for most label propagation algorithms. Like the Louvain method, we start by assigning each vertex its unique label indicating its community affiliation. We then proceed to iterate through all vertices in a random or pseudo-random fashion, and giving the vertex the label shared by most of its neighbors. Ties are uniformly broken at random, however with preference often given to the present label to avoid fluctuations [37]. The procedure is repeated until convergence. This epidemic spreading-like algorithm is remarkably efficient, uncovering a layer of communities fast [34]. The community structure found by this approach is, however, often not the best, usually because one label ends up "flooding" or "plaguing" the majority of the vertices [37]. In recent years several algorithms have spun off the general label propagation algorithm, extending it to increase its accuracy, but often also its complexity.

3.4.2 Hop Attenuation and Vertex Preferences

As noted above, the reason label propagation sometimes outputs less than optimal community structures, is because of the the epidemic nature of the method. Often one can observe large communities holding more than half the vertices in the network. In order to hinder a label to flood the network, the label propagation algorithm above may be extended by adding a score to each label, which decreases

[5]Here, m refers to the number of edges, not the total weight. Label propagation does not work on weighted networks.

as the label travels beyond its starting vertex [24]. Letting $s_j(c^j)$ be the *hop score* of label/community c^j at vertex j, the new community of vertex i, may now be assigned by

$$d^i = \underset{c}{\operatorname{argmax}} \sum_{j \in N(i)} s_j(c^j) f(j)^p A_{ij}, \tag{61}$$

where $f(j)$ is the vertex preference of j. Vertex preference is a function that defines what vertices are given preference in (61) and which are not. For example, we may let the vertex preference of a vertex j be k_j, such that if p is positive, preference is given to high-degree vertices and their communities.

After i has found its new community using (61), the next step is to attenuate the score of label d^i. This is done by giving $s_i(d^i)$ the value of the maximum observed $s_j(c^j)$ in the above equation, and then subtracting a hop attenuation factor δ. The factor δ's role here is to govern how far a label can spread, and may be supplied to the algorithm as an additional parameter to tune the performance. The introduction of δ effectively hinders the formation of huge communities, but also in the cases when the network indeed allows such a structure, and hence high values of δ may hinder healthy growing of communities [24, 37]. The introduction of δ also leaves the algorithm "semi"-supervised, as one often must try different values to find which δ works the best for the network in question. In the next section we'll see how we may avoid the flooding nature of label propagation while dynamically setting δ.

3.4.3 Diffusion and Propagation Algorithm

Vertex preferences and hop attenuation have been carried forward into two unique strategies for label propagation in networks, namely defensive preservation of communities, and offensive expansion of communities. The first give vertex preference to vertices in the core of each community, while the latter give preference to bordering vertices. The two approaches may be combined into what is known as the *Diffusion and Propagation Algorithm* [37].

As noted above, the value of the hop attenuation parameter δ is not easily set without deep prior knowledge about the network, and it may be so that there is no universal value that's good for all networks. Having to manually control this parameter quickly becomes awkward, as we often don't have good knowledge of what the value of this parameter should be. One solution to this is dynamically updating the value of δ after each iteration. In [37] δ is set to be the proportion of vertices that changed their label in the previous iteration, however never letting it exceed $\frac{1}{2}$. When more than half the vertices change label in an iteration, δ is set to zero, to avoid hindering natural community growth.

Defensive and Offensive Propagation

Node preferences was first introduced in [24], and since a variety of different measures including vertex clustering coefficient, degree- and eigenvector centrality, have been used to model preference. Still, no static measure have been found that works for all networks [37]. A proposed solution is to model preference by a random walker on

the network, within each community. Letting the probability that a random walker in community c visits vertex i be p_i, we have that

$$p_i = \sum_{j \in N(i) \cap c} \frac{p_j}{k_j^c}, \tag{62}$$

where k_j^c is the number of edges leaving vertex j for vertices in c. The defensive propagation changes the updating rule in (61) into

$$d^i = \underset{c}{\operatorname{argmax}} \sum_{j \in N(i)} s_j(c^j) p_j A_{ij}, \tag{63}$$

effectively giving preference to vertices in the center of each community, as these vertices have high values of p. Replacing p_i by $(1 - p_i)$ in (63), and replacing k_j^c by k_j in (62) yields the offensive version, which actively gives preference to vertices at the edge of each community. The defensive version unveils a larger number of community cores, which survives throughout the algorithm. The offensive version allows for propagating a label further, and such outputs community structures with a more heterogeneous selection of community sizes. In fact, laying the pressure on the border of the communities expands only the communities that are strongly defined in the network topology, hence resulting in a more natural partitioning than defensive propagation [37].

Basic Diffusion and Propagation Algorithm

The *Basic Diffusion and Propagation Algorithm* is put together by combining the defensive and the offensive label propagation in the following way. First, defensive label propagation is run on the network, producing "estimates" of the community cores. Then all border vertices are relabeled with a unique label and offensive propagation is run. This combined strategy preserves the advantages of both the mentioned propagation types. However, for larger networks the offensive propagation will often yield a very large community, a problem which is attempted solved in the *Diffusion and Propagation Algorithm*.

Diffusion and Propagation Algorithm

The Diffusion and Propagation Algorithm is the most advanced label propagation method we consider in this thesis. In this method we apply the defensive propagation to the network, and proceed to construct the community network as in Section 3.1.4. Now we run the offensive version on the community network to extract a major community. If the communities found on the community network are better according to some measure, than the output of the first defensive iteration, we translate the labels from the community network onto the original network. If not, we simply keep the labels outputted by the defensive approach. If either of the two results in just a single community, i.e. they put all vertices in the same community, we run the Basic Diffusion and Propagation Algorithm on the network and output the structure found. However, if this is not the case, we extract the largest community, $c_{\max}$ (in terms of number of vertices in the original network) and recursively apply the Diffusion and Propagation Algorithm on this subset of the

vertices. After each recursive application, we get a community structure that is a refinement of $c_{\max}$. If the refinement is better (again, by some measure) than $c_{\max}$, we translate the labels found into our full network. For step by step instructions, see Algorithm 3.

Algorithm 3 Diffusion and Propagation Algorithm

1: Run defensive label propagation on the network G to obtain the community structure C.
2: Construct the community network G_C, and run offensive label propagation on it to obtain C'.
3: If the community structure C' is better than C, translate the labels onto G (from G_C), and let C denote the better community structure (regardless of our choice).
4: If there is only one community in C, run the Basic Diffusion and Propagation Algorithm, and *do not continue*.
5: Else, extract the largest community in C, c, and let G_c be the subgraph of G defined on the vertices of c.
6: Run the Diffusion and Propagation Algorithm recursively on G_c to obtain C_c, the refinement of c.
7: If splitting c into the communities in C_c is better than just having c, translate C_c onto G, such that c now has been split, but the other communities in C are intact.

4 Implementation

The methods discussed in Section 2 have been implemented in Python 2.7 [25] for this Master's thesis, and are available on Github[6] and in Appendix B. The methods were implemented as a learning experience, and as well in order to be able to fully control the output and running environment of the methods. Using the original implementations of the Louvain method and the Diffusion and Propagation algorithm to produce the results in Section 6 is unfeasible, simply because of their limited import and output capabilities.

The choice of Python over any other language comes down to preference. Some of the good things are its readability, the fact that it's open source, and its huge user community contributing to free and open packages. In many ways, using Python with NumPy [17] is also extremely similar to the experience of MATLAB [15] or Octave [16], and has only a few syntactical differences [1]. We also wanted to show with the implementation that it is indeed possible to write performance critical software in Python, if you do it the "right" way, and not seek to imitate some other language.

All methods are built on top of a common object-oriented framework for representing communities, allowing for easy interaction with communities and modification of vertex affiliations, see Section A.2. Internally, the graphs are represented

[6] https://github.com/mewwts/communitydetection

by $n \times n$ adjacency matrices, stored using SciPy's Compressed Sparse Row implementation, see Appendix A.1. SciPy's implementation has fast row slice operations and fast matrix-products, which are handy when implementing the methods found in this thesis.

The methods are run from your terminal, and runs on Unix and non-Unix platforms alike. More information about the implementation and the structure can be found in Appendix A. The most important parts of the source code are available in Appendix B.

4.1 Louvain Method

The original implementation of the Louvain method is written in C++ and may be found online[7]. There are few differences in the implementations, apart from choice of programming language, but one of them is how the modularity is calculated.

In the original C++ version the modularity gain associated with moving a vertex from its community to its alternative is only correct up to a constant, in what seems to be a scheme to avoid division by m for the sake of code clarity. While the community associated with the highest gain is still the same, the modularity gain can't be used further in any calculations. The implementation resorts to calculating the global modularity of Equation (7) after each repetition of the first phase. In the Python implementation the gain for each move is added to the initial modularity along the way, avoiding the heavy, global modularity calculation. In fact, a global calculation is only needed once, before the iterations start, but then only over the single-vertex communities, which is simply a $\mathcal{O}(n)$ operation. It is worth mentioning that calculating the global modularity in the original implementation is not as time consuming as it is in the Python implementation because of different memory utilization and because of the languages' obvious performance difference, but it is nevertheless a redundant operation, which does not belong in an efficient implementation.

4.1.1 Calculating the Modularity Gain

The calculation of the gain of moving vertex i from its community d into the community c. The gain is calculated as in (36), and as noted in Section 3.1.3 the move only affects the two communities involved, so a result the modularity gain is quite easy to compute. It is implemented in Listing 7 in Appendix B. A simple for-loop is used to iterate through the neighbors of the vertex we consider, calculating the q_{ic} and q_{id} of (36) as it goes. As we remember from (30), q_{ic} defined as the sum of (28) for each vertex in c. So, when we iterate over the neighbors j of vertex i, we add the quantity q_{ij} to q_{ic_j} where c_j is the community of j. This way, we calculate q_{ic} for all neighboring communities in one for-loop, and if we're feeling particularly effective, we may even keep track of the community c whose q_c is the largest during the for-loop, avoiding iterating through all the different q_{ic} to find the maximum.

[7] https://sites.google.com/site/findcommunities/

4.2 Community-Dissolve

More interesting is the implementation of community dissolve, as it faces several challenges. The Community-dissolve Algorithm utilizes a subclass of the community-object mentioned above, which at any time holds the correct modularity of each community. The global modularity may then be found simply as a sum over these values for each community. As the moving criterion in Community-dissolve is that the total gain in modularity from moving all vertices from a community is larger than the original communitys modularity, it's critical that this quantity is calculated correctly. The code that calculates this is the function `mass_modularity` found in Listing 7. In reality it is just an extension of the calculation of modularity gains for a single vertex, but for the modularity gain to be correct, we must remember that two vertices x and y, going to the same community has a joint modularity even if there is no edge (x, y) in the edge set, E. This means that the calculation of the modularity gains for a community is heavier than just a single iteration over the edges, as one must also iterate over the vertex pairs that share no edges.

4.3 Degree-Rank

The initial idea behind the Degree-rank Algorithm was to use our prior knowledge of vertex degree as a bias of a Louvain-like iteration, in order to speed up the convergence. What was done instead was using the sorted vertex degrees the order of iteration in a stand-alone community detection method. In practice, the Degree-rank algorithm is also slower than the Louvain method on larger networks. One of the reasons is that sorting the vertices by their degree, is in the best case a $\mathcal{O}(n \log n)$ operation. More notably, as will become apparent in Section 6, the method provides slightly poor results.

4.4 When to Stop Iterating

The inner loops of the algorithms defined above in Sections 3.1, 3.2 and 3.3 returns when the last iteration through the vertex set did not yield any gain in modularity higher than a provided threshold. The default value for this threshold is 0.02, but it may be set for each run. The outer loop of these algorithms may be seen in Listing 2, and it handles constructing the community network after each of the algorithms' first phase, and it returns when the last iteration yielded no gain in modularity.

4.5 Diffusion and Propagation Algorithm

Label propagation is in its essence a lightening fast framework for community detection. In [37] the Diffusion and Propagation Algorithm is said to have close to linear complexity, in fact it's measured to $\mathcal{O}(m^{1.19})$, which should scale better than the basic label propagation algorithm. Throughout the Java implementation that follows the paper [37], the measure used to determine whether a community structure is better than an other, for example when deciding to keep the labels of the constructed community network or not, is modularity. This is unfortunate for a few reasons. First, the modularity is not calculated while propagating labels, leaving the

heavy calculation of (7) to be done several times during the course of the algorithm. While this is done in the original C++ implementation of the Louvain method as well, as noted in Section 4.1, such a calculation is unnecessary and very time consuming in Python, especially for larger networks. Second it's slightly inappropriate for a method alternative to modularity maximization to use modularity as a measure of "goodness", especially when it sets the method back in terms of performance.

The Python implementation in Listing 6 follows the Java code closely, deviating mostly by utilizing custom data structures to store communities. The reason for this is that the Java code provided deviates significantly from the pseudo-code in [37], and hence was hard to follow. This makes the Python implementation very slow, and it fails to converge in 24 hours on the real world datasets with a few million vertices.

A small fix that would help speed up both implementations of the algorithm follows. When determining if the refinement of some community c increases the global modularity, it is unnecessary to calculate the modularity for the whole network. It is enough to calculate the modularity of community c, Q_c, and compare it against the sum of the modularities of the refinements. That is, we include the refinement, C_c, only if $\sum_{c' \in C_c} Q_{c'} > Q_c$. This would speed up each recursive application and in principle make the Diffusion and Propagation Algorithm run in a comparable amount of time to the Louvain method.

4.6 Random Numbers

The Louvain method, Community-dissolve and the Diffusion and Propagation algorithm should visit the vertices in their main loop in a random order to avoid getting stuck in local optima. In practice this means drawing numbers from $\{0, 1, \ldots n-1\}$ without replacing the numbers that are drawn. This can be done by storing the sequence as a set, converting it to a list and using some random seed to choose elements, then removing the item from the set. However, as this is at least an $\mathcal{O}(n)$ operation, that has to be done for each step in a for loop of length n, our methods would be of quadratic complexity $\mathcal{O}(n^2)$. The better way would be to shuffle the list in place before the iteration starts. This again, is an $\mathcal{O}(n)$ operation, but it turns out we can avoid it and obtain sufficient randomness by doing something less obvious. Given any number p less than n such that p and n are relatively prime, i.e. $\gcd(p, n) = 1$, we can generate all the numbers from 0 through $n-1$ by a simple multiplication and modulo scheme. Any number in $\{1, \ldots n\}$ multiplied by p modulo n will produce another number in $\{0, \ldots, n-1\}$. So if we for each iteration of the above mentioned community detection methods, take the number of the current iteration, multiply it by p and modulo it by n, we will get some number from $\{0, \ldots, n-1\}$. What we're really doing is just generating the group $\mathbb{Z}/n\mathbb{Z}$, and strictly speaking it's not really a random order as for each p there is a defined order of the returned vertices. However, p is chosen at random, and if p does not coincide with some hidden numbering of the vertices in our graph, the vertices will be drawn in what seems like a random way. The code can be found in Listing 12, and the calculation of gcd is managed by an implementation of the Binary-gcd Algorithm [36].

4.7 Testing Infrastructure

The testing interface is a Python program that is run in your terminal. You simply feed it a directory from which it crawls all subdirectories looking for files. For each directory it appends a tuple ([file1, file2, ...], filegt), where the first element is a list of filepaths to run each community detection method on, and the second is the path to the ground-truth community structure. The module uses the Python Multiprocessing module, applying jobs asynchronously to all processors in your computer. The results in Section 6 are obtained by running some 12000 tests on a 24 core computer, taking only an hour to finish. The code may be inspected in Listing 15.

5 Datasets

5.1 Benchmark Networks for Community Detection

In Section 6 we'll thoroughly test and benchmark the methods for community detection presented in Section 3 on networks with a known community structure. In order to test in a sound way, we need a wide array of test networks, with different properties. This means that we have to generate such graphs, and below we review two known classes of benchmark graphs.

5.1.1 Girvan-Newman Benchmark

Girvan and Newman introduced in a paper [14] a class of computer generated graphs for benchmarking community-detection algorithms, which since has been widely adopted and used as a measure of performance. The graphs are generated with 128 vertices divided into four communities consisting of 32 vertices each. Further, edges are placed independently at random, with two different probabilities P_{in} and P_{out}. Vertex pairs within a community have edges placed between them with the probability P_{in}, while vertices in different communities are connected with probability P_{out}. Of course, if we are to keep our notion of a community that vertices within communities are more tightly connected than expected, we need to impose that $P_{in} > P_{out}$. It is common to choose P_{in} and P_{out} such that the average degree of a vertex, $\bar{k}$, is approximately 16.

5.1.2 LFR Benchmark

The LFR benchmark [22, 21, 20], introduced by Lancichinetti, Fortunato and Radicchi, seeks to provide a class of more realistic benchmark graphs which models two important properties of real-life networks; the heterogeneity of vertex degrees and community sizes [30, 9]. The benchmark is an extension of the Girvan-Newman benchmark. We will use two versions of this benchmark. Both generates undirected graphs, but one makes unweighted graphs while the other makes weighted. In both versions the vertex degrees are distributed by a power law with exponent τ_1, and community sizes follow a power law with exponent τ_2. The parameter μ_t is the average topological mixing parameter, which specifies the average ratio between the number of edges leaving a vertex reaching vertices outside its community, and the

number of edges leaving the vertex. The mixing parameter effectively determines how significant the communities are. As we'll see in Section 6, a value of μ_t larger than 0.5, makes it very hard for the algorithms to find the communities within in the network. An additional parameter μ_w is used for the weighted networks, and it expresses the average ratio between the weight from a vertex to vertices outside of its community, and the degree of the vertex. When testing the algorithms on the unweighted networks, we'll keep the network size and both τ_1 and τ_2 fixed, while varying the mixing parameter μ_t in the interval $(0.2, 0.9)$ in order to determine how the algorithms perform in finding communities that are increasingly indefinite. In the weighted case, we'll also keep μ_t fixed while varying μ_w, as well as testing the case where $\mu_t = \mu_w$.

5.1.3 Test Network Parameters

For the LFR-benchmark graphs, we follow for the most part the parameters used in [21], for easier validation and comparison. For both classes of graphs, weighted and unweighted, we generate sets of networks with 1000 and 5000 vertices. We let the average and maximum degree equal to 20 and 50, respectively. For the largest networks with 5000 vertices, we let the minimum community size be 20 vertices, and the maximum 100. For the smaller networks we let the minimum community size be 10 and the maximum 50. We generate unweighted networks for values of μ_t in $(0.2, 0.3, \ldots 0.9)$, omitting the edge cases 0.0, 0.1 and 1.0 as they are uninteresting. For values below 0.2, the community structure should be easy to reveal, and for 1.0 it should hardly be present in the network. For the weighted case we fix the topological mixing parameter μ_t first to 0.5, and vary μ_w within the above interval, then repeat for $\mu_t = 0.8$. We also include the results when we fix $\mu_t = \mu_w$ and let them vary within $(0.2, \ldots 0.9)$.

5.2 Telenor Datasets

We also benchmark the algorithms on two datasets provided by Telenor Research. Our goal is to investigate whether or not the Telenor customer base admits a significant community structure. Both datasets are adjacency matrices representing graphs with some 2.9 million vertices, where each vertex characterize a phone number. The first data set, which we'll call the "call-graph", has directed edges representing phone calls from one vertex to another. The weight on the edge uv is a real number representing the number of minutes u has called v. The other data set has edges representing communication via SMSs between vertices. The weight on the edge uv is a positive integer, representing the number of SMSs sent from u to v. We call this data set the "SMS-graph".

5.3 Pre-processing

Both datasets include self-loops, which indicates that a person has called or messaged him- or herself. We consider this as noise, which we remove, as self-calls or messages are often mistakes or notes to oneself. In addition, these self-edges may hinder

the formation of communities, and as such, we remove them simply by setting the diagonal of the adjacency matrix A, of any of the above mentioned graphs, to zero.

As mentioned, these communication networks are directed, but the methods discussed in this paper, and most other community detection approaches, do not work on directed networks, as this might require a generalization of the modularity function [23]. Although neglecting the information encoded in directed networks may seem like a big sacrifice [21], we argue below that it's fine to do so in communication networks, like the Telenor datasets.

5.3.1 Symmetrization

We'll consider two different ways to symmetrize a matrix A. The first and easiest way is simply letting $A^* = \frac{1}{2}\left(A + A^T\right)$, letting the ij'th entry of A^* be the mean of the ij and ji elements of A. The alternative is based on the notion of reciprocal ties, letting $A^*_{ij} = A^*_{ji}$ be nonzero only if both A_{ij} and A_{ji} are nonzero. This is done by letting $A^*_{ij} = A^*_{ji} = A_{ij} + A_{ji}$ if both of these two entries of A are different from zero.

Let us now consider the two approaches. The "mean"-approach doesn't care if j called i, as long as i called j. The reciprocal ties approach does, and as it turns out, not caring about reciprocity can be bad. The networks provided by Telenor are communication networks, of all Telenor customers, including call-centers and and tele-marketers. We have no way of knowing which vertices are which, as the network is anonymous, but we may have a hunch that the vertices with the highest number of outgoing or ingoing edges may be one of the two. Now, if we simply symmetrize by the first approach, these vertices with high out- or in-degree would have a large degree in the symmetric A^*, as well. If we symmetrize by reciprocal ties, however, the vertices with high in- or high out-degree in A, will have low degrees in A^*. Leaving us with less of a negative impact on the community structure, caused by these "directed" vertices. We may use this argument to justify the symmetrization in the first place[8].

5.3.2 Connectivity

We saw in Section 2.2 what it means for a graph to be connected, and usually we assume that this property is present in the input graphs of our community detection algorithms. Naturally, if the graph is disconnected, each connected component will give rise to a community structure of its own, as no vertex in the component is connected to a vertex outside it. This is fine, and we may just as well include them in the analysis. However, as an illustrating example, the raw "call-graph" has more than 400,000 connected components. Most of them are of size 1 or 2, and hence they, trivially, form communities of their own. We aren't really interested in revealing communities that are already defined in the topology, as they could be revealed by simply using a breadth-first search. More interesting are the groupings of vertices that are not apparent. With this in mind, we choose to only consider the largest connected component of the graphs. The largest connected component of a

[8]Not that we have a choice, since the methods really can't handle directed networks

graph $G = (V, E)$ is the connected subset V_C holding the most vertices. The call- and SMS graph has approximately $2,000,000$ and $1,900,000$ vertices, respectively.

5.4 Modifying the Adjacency Matrix

Almost any matrix can be thought to represent some underlying graph. If we run our community detection algorithm on some matrix representing a slightly different graph, G', than our original graph, it happens that we can learn something about the community structure in $G = (V, E)$. In this section we'll see what happens when we leave our vertex set V intact, while modifying our edge set E by some matrix transformations.

5.4.1 Matrix Powers

Let A denote the adjacency matrix of some graph G whose community structure we're interested in revealing. A well known fact in graph theory, is that the number of walks of length l between vertices u and v in G is equal to the uv'th entry of the matrix A^l, A^l_{uv}. Recall the definition of modularity in Equation (6). The idea of the equation is that communities have more than the expected number of edges between its vertices. Changing the matrix used in (6), from A to A^l, we discover a concept known as "walk-modularity" [27]. If we denote the walk-modularity by Q_l, we may define it as

$$Q_l = \frac{1}{2m_l} \sum_{ij} \left[A^l_{ij} - \frac{k^l_i k^l_j}{2m_l} \right] \delta_{c_i c_j}, \tag{64}$$

where k^l_j is the degree of vertex j in A^l, and not the l'th power of k_j, as to follow the null models proposed earlier in Section 2.4.2. Walk-modularity is based on the notion that a community will have higher than expected number of walks between its vertices. Simply feeding our algorithms A^l instead of A is enough to maximize this walk-modularity. Since the vertex set of the graphs are the same, the communities found using the walk-modularity are valid in G.

5.4.2 Matrix Exponential

Consider the matrix function e^A, and its power series expansion

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \ldots + \frac{A^k}{k!} + \ldots = \sum_{i=0}^{\infty} \frac{A^i}{i!}. \tag{65}$$

If we disregard the denominator, we interpret the ij'th entry of e^A to be the sum of all walks of all lengths from i to j. Now, considering the the denominator as well, we see that the $\frac{1}{k!}$ is a penalty factor for any walk of length k, deeming shorter walks more important than long walks in e^A. The ii'th entry of the matrix, e^A_{ii}, is called the *subgraph centrality* of vertex i [12], whereas e^A_{ij} is called the *subgraph communicability* between vertex i and j [4]. *Centrality*, as a concept, measures a vertex' relative importance in the graph. As such, a vertex with high subgraph centrality, relative to the others, is considered to be more important than vertices with low subgraph

centrality. If the ij'th entry of e^A is high relative to other entries, this indicates that information flows more easily between i and j than between vertices with lower communicability [4, 12]. When we try to reveal communities in e^A, we effectively try to identify groups of vertices that communicate better with each other than one would expect.

5.4.3 Edge Restriction

All of the above described matrix functions will in general produce matrices denser than the original matrix. For example; in the second power of A, A^2, the column indices of the nonzero entries on row i are the vertex i's neighbors neighbors. Or, in other words, vertices reachable by following two consecutive edges away from i. In the third power, a nonzero entry on row i indicates that a vertex is reachable in three steps from i. The methods for community detection in this thesis all have complexities dependent of the number of edges, which obviously lead to slow convergence times if run on a dense matrix. We also note that the funtion $exp(A)$ leaves a fully dense matrix with $m = n \times n$ entries. In such a dense graph, grouping vertices into meaningful communities becomes very hard, as any given vertex will have $n-1$ neighbors.

We here propose a way around this, *edge restriction*, which on a graph G means that we switch the weight function of G, ω, with the weight function, ω' of some other graph G', restricting it to the edge set of G. In our case, the weight function we're interested in is the weight function resulting from the above mentioned matrix transformations. Alternatively, we may think of it as keeping the weight function of our new graph, but we consider only the edge set of the original graph. In a matrix, this is equivalent to considering the entry A'_{ij} of our new matrix A' only if the corresponding entry of A is different from zero, that is $A_{ij} \neq 0$.

A problem with this approach is that if the vertex u is connected to vertex v by an edge, but the two do not share any neighbors, $|N(u) \cap N(v)| = 0$, the uv'th entry of A^2 will be zero. This follows from the fact that if u and v do not share any neighbors, there are no ways to get from u to v in exactly two steps, and hence this entry will be zero in A^2. The worst case scenario is that the edge restriction applied to this network leaves the resulting graph disconnected. To overcome this, we instead consider $\hat{A}^2 = (A + I)^2$, adding self-loops to every vertex in A before squaring the matrix. Now, any vertex that is reachable from u in one step in A, is reachable in two steps in $A + I$, resulting in a nonzero uv'th entry in $\hat{A}^2$, even if u and v share no neighbors.

6 Results

6.1 Validation

We start by running the algorithms on a few well known datasets, to validate our implementations' output, and to ensure that the novel approaches Community-dissolve and Degree-rank are viable methods. In Table 1 the best values of modularity obtained over 10 runs on a few datasets, are presented. As expected, Community-

Table 1: Modularity obtained on a few networks. The best value out of 10 runs is reported. The values in parenthesis are modularity obtained with the Java version of the Diffusion and Propagation Algorithm.

	Louvain	Dissolve	Rank	Diff. & Prop.
Karate [39]	0.4198	0.4198	0.3875	0.4156 (0.416)
Lesmis [18]	0.5600	0.5600	0.4998	0.5519 (0.553)
Political books [19]	0.5268	0.5268	0.5216	0.5238 (0.524)

dissolve is performing level with the Louvain method, but somewhat disappointingly it does not provide better results. Degree-rank is struggling to obtain a high value of modularity, and is not really performing anywhere near the state-of-the-art. The Diffusion and Propagation algorithm is also slightly behind both the results of the Louvain method, but also the values reported in [37]. The values obtained by the original implementation (run on the same machine as the Python implementation), are provided as well and also deviate slightly from the previously reported values. The Python implementation of the Louvain method was validated and tested against the original in [33].

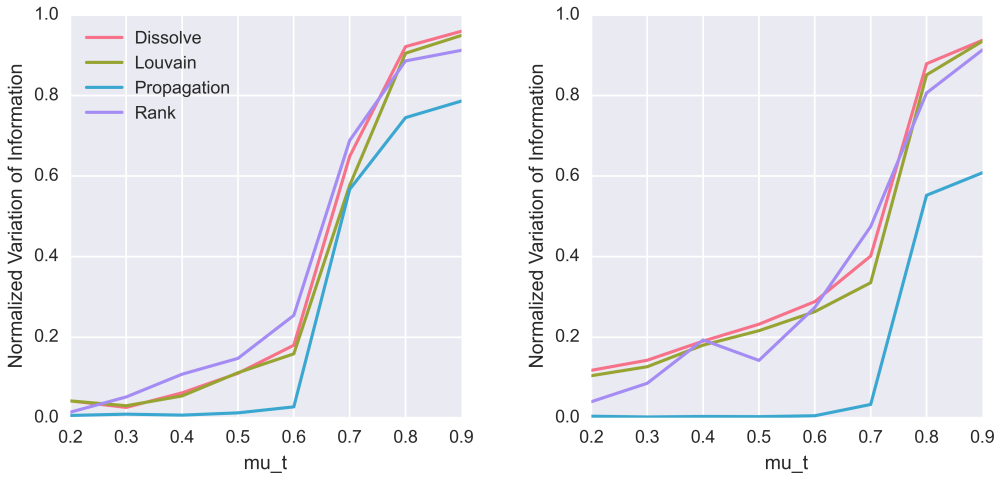
6.2 Benchmarks and Comparison of Community Detection Methods

In this section we'll thoroughly review the performance of the Louvain method, the Diffusion and Propagation Algorithm, Community Dissolve and Degree-rank on the LFR-benchmark, presented in Section 5.1.2. The methods are run on both weighted and unweighted networks, with parameters as specified in Section 5.1.3. For the methods that output a hierarchical community structure, that is the Louvain method, Community-dissolve and Degree-rank, the results are obtained by using the top level of the hierarchy, if no other remarks are made.

6.2.1 Unweighted Networks

Figure 6 shows the performance of the methods on unweighted networks of sizes $n = 1000$ and $n = 5000$. The results are given as the joint entropy-normalization of the variation of information, as shown in Section 2.5.5. We will use this measure throughout the section, and we refer to this metric simply as normalized variation of information, or even NVI. As NVI measures the distance between two clusterings, where in our case one of them is the gold standard/ground truth, lower values are always better.

We notice how the Louvain method and the other modularity-based methods are performing worse than the label propagation algorithm. While the Diffusion and Propagation algorithm succeeds in revealing the full ground truth partitioning up to $\mu_w = 0.6$ for $n = 5000$, the Louvain method fails to find the ground truth even when $\mu_w = 0.1$. In Figure 6, the top level of the outputted hierarchical community structure was picked for each test network. If we instead consider the level in the hierarchy that minimizes the NVI, we get the results presented in Figure 7. It tells



(a) $n = 1000$ (b) $n = 5000$

Figure 6: Comparison of all methods on unweighted LFR-benchmark graphs. Top level of hierarchy chosen.

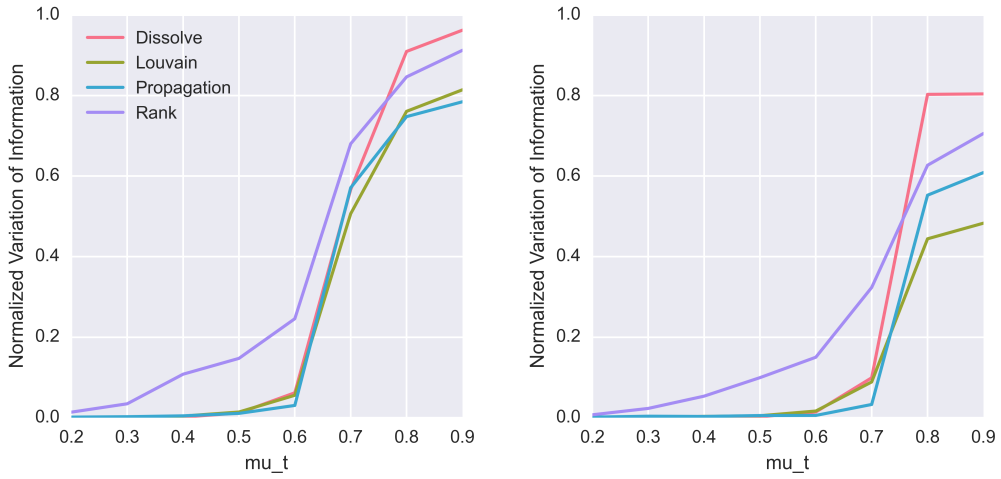
a different story, where the Louvain method is even better than the Diffusion and Propagation Algorithm, at least for $n = 5000$. We must, however, keep in mind that it is the last level in the hierarchy that is the "final" results of the modularity based methods, and hence it might be misleading to consider Figure 7. From both figures, we see that detecting communities when the fraction of inter-community edges, μ_t increases beyond 0.5 is hard, and there are significant and rapid ascents present in both Figures 6a and 6b.

Nevertheless, by further inspection, we see that all methods perform better on the higher network size when we pick the NVI-minimizing level. While this could simply be a result of the network size, most likely it is a by-product of increasing the community sizes. Modularity has been shown to suffer under the *resolution limit* [13], which in essence means that it is biased towards detecting larger communities, rather than smaller ones.

6.2.2 Weighted Networks

When benchmarking the algorithms on weighted networks, we fix the topological mixing parameter μ_t and vary the mixing parameter for the weights μ_w. This means that for a given set of networks with the same μ_t, the number of intra-community edges will be the same, but the fraction of edge weights leaving a vertex to vertices outside the community will vary with μ_w. In Figures 8 and 9 we have plotted the results of benchmarking the Louvain method, Community-dissolve and Degree-rank on weighted networks of size $n = 1000$ and $n = 5000$, respectively. In Figure 9, that is for $n = 5000$, the case when $\mu_t = \mu_w$ is also included. In this last case, the Diffusion and Propagation is also tested, even though it turns the weighted graphs into unweighted ones. This is because we may compare the results of the other methods against it, although it's performance obviously only depends on μ_t.

Looking at Figure 8 we see evidence of the same trends we saw for the unweighted



(a) $n = 1000$ (b) $n = 5000$

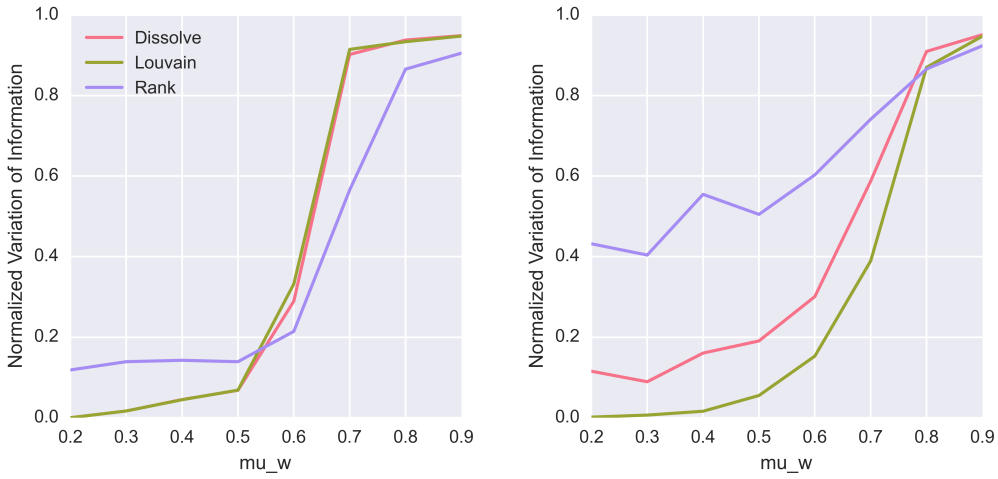
Figure 7: Comparison of all methods on unweighted LFR-benchmark graphs. Minimal NVI-resulting level chose.

networks in Section 6.2.1. In Figure 8a, Community-dissolve is very similar to its parent, the Louvain method. When we increase the fraction of inter-community edges from 0.5 to 0.8, however, Community-dissolve apparently makes bad choices compared to the Louvain method, as can be seen in Figure 8b. The Louvain methods ascent in this plot comes both later, and less steeply now than for $\mu_t = 0.5$, effectively providing better recovery of the ground truth partitioning. The reason behind this is not clear, as it is quite unintuitive that the partitioning revealed by the Louvain method is better when the intrinsic partitioning is less defined in the topology.

When the fraction of edges between communities is equal to the fraction of edges within communities, as seen in Figures 8a and 9a, Degree-rank provides a better recovery of the ground truth than both Louvain and Community-dissolve, for values of μ_t above 0.5. However, we see that both Community-dissolve and Degree-rank are especially sensitive to changes in the topology that makes communities less defined, i.e. the parameter μ_t, as their performance is much worse when μ_t changes to 0.8, see Figures 8b and 9b.

The LFR-benchmark graphs do not provide particularly heterogeneous degree sequences. Its maximum degree is set to be 50, and its average to 20. In real world networks, the maximum degree in a network of size $n = 5000$, would typically be much larger. In a network of size $n = 1000$, such a degree can be justified if the network is sparse, and we see in the plots that the method performs better in this case. As the Degree-rank method is tailored to walk down a heterogeneous degree distribution, we can understand that it stumbles when it meets the more homogeneous networks for $n = 5000$.

In Figure 9c the topological mixing parameter and the mixing parameter are set to the same values. However, here we again see that the Diffusion and Propagation algorithm reveals the complete gold standard partitioning all the way up to $\mu_t = \mu_w = 0.6$. The other methods fail to perform this well, even though they have the extra information that the label propagation approach lacks; the edge weights.



(a) $\mu_t = 0.5$ (b) $\mu_t = 0.8$

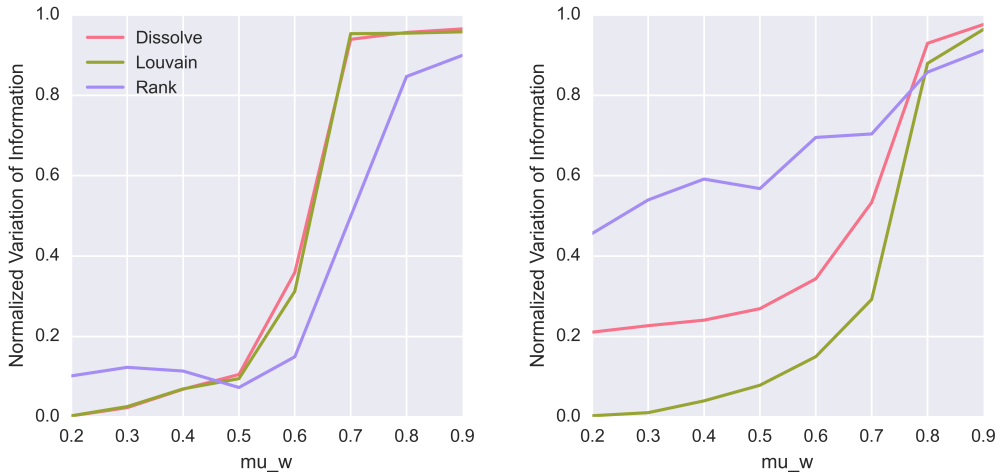
Figure 8: Comparison of all methods on weighted LFR-benchmark graphs of size $n = 1000$.

For high values of the mixing parameters μ_t and μ_w, recovering the planted ground truth partitioning becomes almost impossible. The algorithms all output community structure that are almost entirely different from the planted one. In these cases, how well are the communities really defined? For $\mu_t > 0.5$ the fraction of edges within communities decrease. When μ_t approaches 1, there are hardly any edges within communities. Are the communities really defined at this point? The same goes for μ_w. When we shift all the weight onto edges between communities, how can we expect any methods to reveal the planted partitioning?

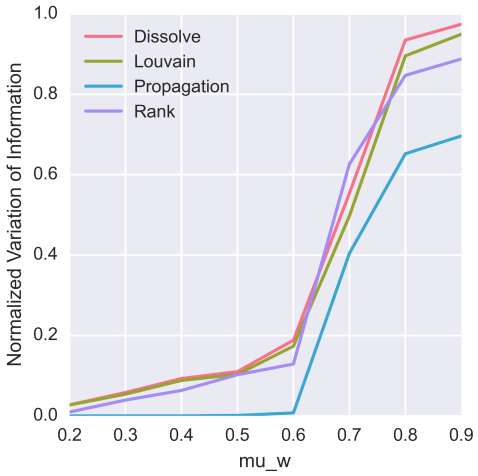
6.3 Effects of Network Structure Alterations

In this section we examine how altering the underlying network of the community detection algorithms affects the results. As we saw in Sec. 5.4 there are three different matrix transformations we wish to investigate; the second power of A, A^2, the third power A^3, and the matrix exponential $exp(A)$. First we'll investigate when A is representing an unweighted network. The entries of A are then either 0 or 1, but the entries in A^2 and A^3 are integers not necessarily equal to 1. We would like to point out that the matrices used in this section really are $\hat{A}^2$ and $\hat{A}^3$, which we presented in Section 5.4.3. The entries in the matrix exponential are real numbers. As noted above in Section 6.2, the Diffusion and Propagation Algorithm will always replace the edge function a graph with the identity, and together with the edge restriction, it will always be run on the same graph. We therefore omit any results including the label propagation approach here. In addition, the results of altering the adjacency matrix for Community-dissolve and Degree-rank are omitted. This is because the outcomes are completely analogous for these methods as for the state-of-the-art Louvain method.

Consider the results altering unweighted networks, as presented in Figure 10. We see indications that the second and third power of A are useful when detecting

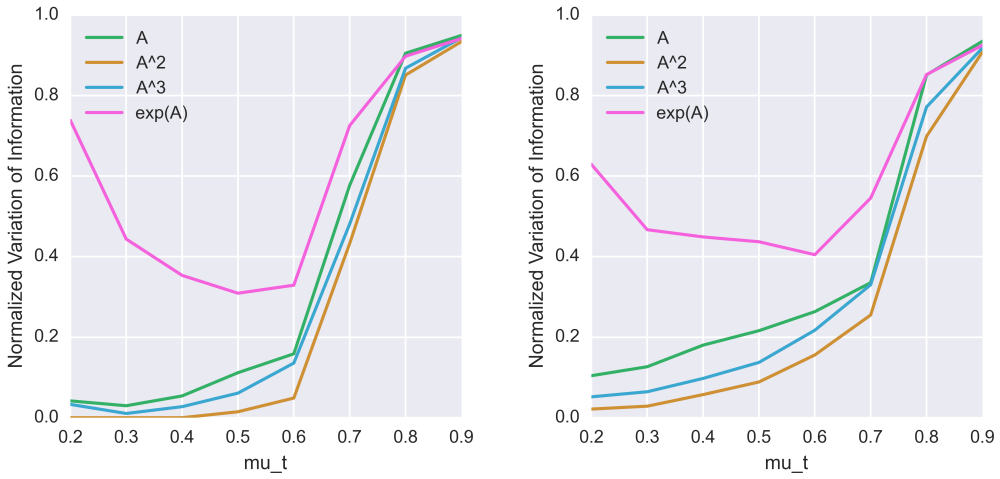


(a) $\mu_t = 0.5$ (b) $\mu_t = 0.8$



(c) $\mu_t = \mu_w$

Figure 9: Comparison of the methods on weighted LFR-benchmark graphs of size $n = 5000$.



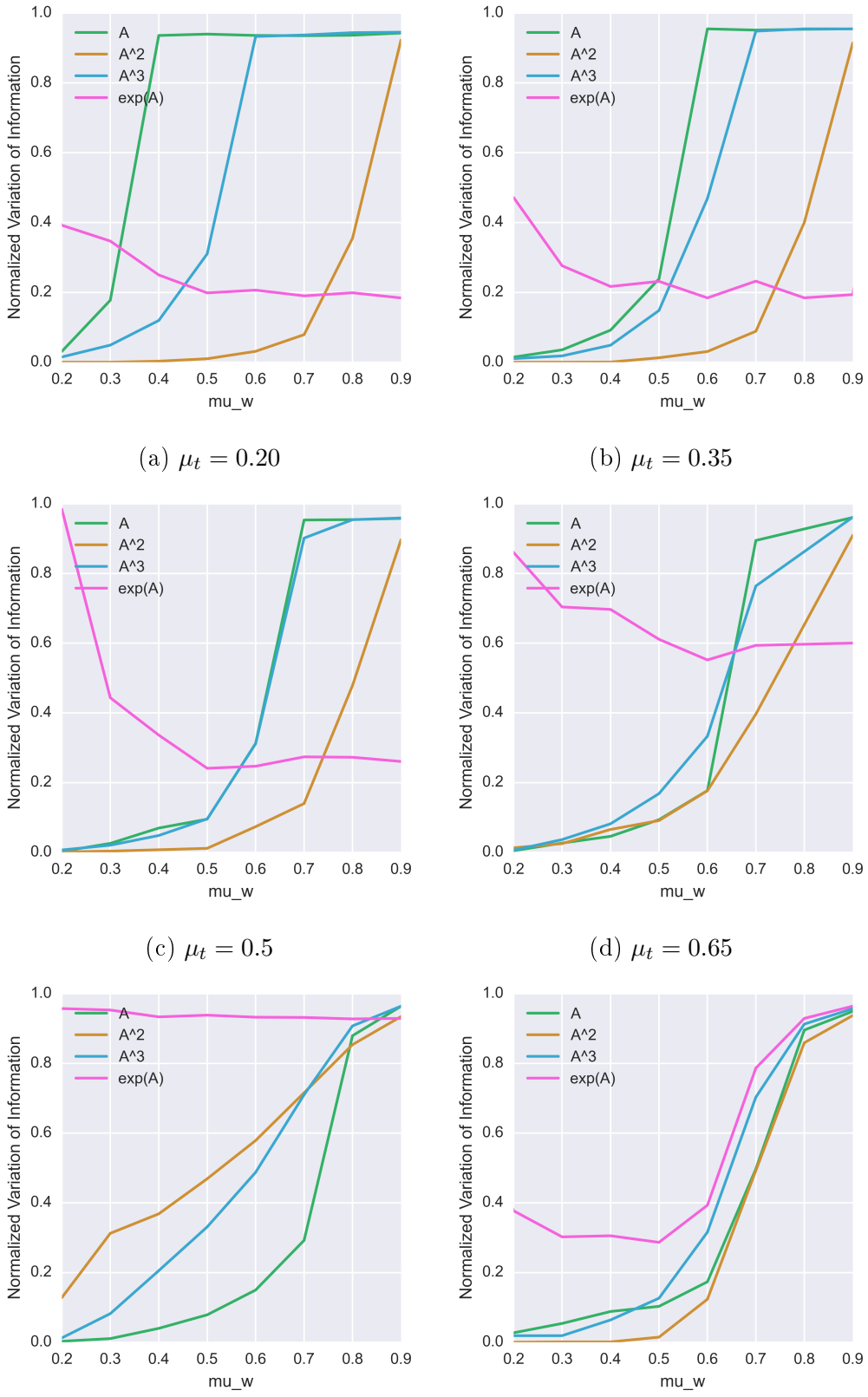
(a) $n = 1000$ (b) $n = 5000$

Figure 10: Influence of matrix transformations on unweighted LFR-benchmark graphs for the Louvain method

communities, each of them helping the Louvain method to unveil a slightly more correct community structure. On the unweighted networks, it is clear that the matrix exponential is not helping at all, worsening the performance significantly over all values of $\mu_t < 0.8$.

For the weighted case, we only examine the networks of size $n = 5000$, but the again for a broader array of values for μ_t than before. Figures 11a to 11e shows the effect of transforming the weighted networks for values μ_t equal to 0.2, 0.35, 0.5, 0.65 and 0.8. The results are remarkable. Transforming the matrices by the transformation e^A, gives results that are seemingly unaffected by the distribution of weights in the graphs. If anything, the Louvain method applied to e^A performs better when the weights are placed between communities, than within. The results of e^A are on the other hand very susceptible to changes in topology, moving from a consistent "belt" around 0.2 when $\mu_t = 0.2$, to a belt around 0.9 when $\mu_t = 0.8$. As we remember from Section 5.4.2, when detecting communities in e^A, we are revealing groups of nodes that communicates better with each other, than with the rest of the network. This is done by summing the entries of A, $\frac{A^2}{2!}$, $\frac{A^3}{3!}$ and so on. As noted before, the ij'th entry of A^x, where x is some positive integer, is the number of walks (the weight of all walks) of length x. In order to find communities in a weighted matrix A, clearly it is important that vertices within communities have strong connections in terms of edge weights. However, in A^2, it is more important that two vertices share many of the same connections, as the weights of all these entries are summed. The same goes for A^3. This explains why the topology is more important than edge weights for the matrix exponential.

Another striking result from Figure 11, is the performance of A^2. When communities are strongly defined in the topology, that is for values of $\mu_t < 0.5$, detecting communities using A^2 provided partitions much more similar to the ground truth than the partitions outputted while using A. This is even true for $\mu_t <= 0.65$. We again point out that the results here are obtained using the top level of the hierarchy,



(a) $\mu_t = 0.20$

(b) $\mu_t = 0.35$

(c) $\mu_t = 0.5$

(d) $\mu_t = 0.65$

(e) $\mu_t = 0.80$

(f) $\mu_t = \mu_w$

Figure 11: Influence of matrix transformations on weighted, $n = 5000$ LFR-benchmark graphs for the Louvain method.

Table 2: Modularity and number of communities on the Telenor call dataset. The run with the best modularity out of 5 is reported here.

	Louvain		Degree-rank		Community–dissolve	
Pass	n_C	Q	n_C	Q	n_C	Q
1	415540	0.7427	225367	0.8116	343747	0.7685
2	75536	0.8598	25565	0.8816	31254	0.8686
3	8942	0.8903	10131	0.8836	942	0.8900
4	2014	0.8980	8179	0.8838	621	0.8906
5	705	0.8995	8140	0.8838		
6	506	0.8996				
7	498	0.8996				

and the results of using the NVI-minimizing level is a bit better for A and A^3. With that out of the way, the information we may draw from Figure 11, is that, when communities are clearly defined in the topology, but the weights are predominantly present between communities, using A^2 and e^A for community detection may help unveiling the community structure.

Figure 11f depicts the results when μ_t is fixed to μ_w. The results are akin to the unweighted case, seen in Figure 10.

6.4 Telenor Data

We here present the modularities obtained by the different methods on the Telenor Data sets. As the Python implementation of the Diffusion and Propagation algorithm did not converge in 24 hours, there are unfortunately no results from it to present. Typically, the other methods all use under an hour for datasets with 2 million vertices, and the Louvain method below 20 minutes.

In Table 2 the results of the analysis of the call-graphs community structure is presented. The values of modularity here are strikingly high, indicating that the preprocessed call-network admits a significant community structure. The number of communities found in the last pass of the Louvain method is 498, a number that perhaps may be linked to the number of municipals in Norway, namely 428. As stated in the introduction, the community structure combined with meta-data, could provide powerful information about the entities in the network. It's also exciting to see that, despite the bad results obtained in Section 5.1, Community-dissolve and Degree-rank obtains comparable modularities to the output of the Louvain method.

On the SMS-graph, the best obtained modularity is lower than in the call-graph. Here, the Louvain method only obtains a value of 0.8422. Although lower than for the previous graph, we may indeed claim that the SMS-graph also admits a subdivision into significant communities. What's striking about the result on the SMS-graph, however, is the number of communities found by the Louvain method. This number is only 110, which should be a manageable number of communities for visualization in programs such as Gephi [3]. Visualizing community structures are however outside the scope of this thesis, but the community structure of the call-graph has been visualized in [33], but then with a mean-symmetrization of the

Table 3: Modularity and number of communities on the Telenor SMS dataset. The run with the best modularity out of 5 is reported here.

	Louvain		Degree-rank		Community-dissolve	
Pass	n_C	Q	n_C	Q	n_C	Q
1	352043	0.6475	141802	0.7042	266375	0.6634
2	60415	0.7507	4357	0.8303	17645	0.7722
3	2844	0.8262	2795	0.8319	215	0.8241
4	485	0.8398	2570	0.8319	165	0.8253
5	162	0.8421	2567	0.8319	162	0.8253
6	113	0.8422				
7	110	0.8422				

network.

As mentioned above, the Python implementation of the Diffusion and Propagation Algorithm did not converge in comparable time to the other methods. We did however have the chance to run the Java implementation of the Diffusion and Propagation implementation on both data sets to see if we could obtain a better value of the modularity than the Louvain method. After all, the results in Section 5.1 indicates that the label propagation algorithm may better at revealing community structure. The Diffusion and Propagation Algorithm, however, does not obtain values anywhere close to the modularity-based methods. On the call-graph, it found 16247 communities with an associated value of modularity 0.683. The results on the SMS-graph is worse. Here it found 10551 communities obtaining only a modularity of 0.462.

7 Closing Remarks

In this thesis we have benchmarked four community detection methods on computer generated bechmark graphs. The methods considered are the Louvain method and the Diffusion and Propagation Algorithm, as well as the novel approaches Community-dissolve and Degree-rank. We saw in Section 6 that the Diffusion and Propagation Algorithm offers the best performance on unweighted graphs. When considering the highest level in the hierarchical output of the state-of-the-art Louvain method, we see that it often fails to uncover the full ground truth partitioning in both unweighted and weighted networks alike. Both Community-dissolve and Degree-rank have comparable results to the Louvain method for unweighted graphs. For weighted graphs, the novel approaches are more sensitive to changes to the topology than the Louvain method, and have worse performance when the topological mixing parameter μ_t increases.

We questioned in Section 6.2.2 the existence or significance of communities in the extreme cases of μ_t and μ_w, and particularly how we could expect community detection algorithms to uncover the partitioning when it is only weakly defined. However, we saw in Section 6.3 that we are actually able to recover significant parts of the ground truth community structure when μ_w is large, simply by transforming the underlying adjacency matrix, A, by A^2 or e^A. Of course, transforming networks

by these functions are heavy, time consuming operations and are perhaps not viable if speed is the number one priority. We did, however, introduce a technique for reducing the number of edges in the transformed matrices, called edge restriction, in Section 5.4.3. An experiment was done, trying to detect communities in e^A without restricting the edges in this way. Although the results are omitted from this thesis, the unrestricted graphs did not aid the unveiling of the ground truth partitioning at all, they only increased the convergence times for all methods. From this we understand that edge restriction is a viable tool in community detection, that should be explored further. More study should also be made one the influence of matrix transformations. For example, is there a matrix transformation that can aid community detection algorithms when μ_t increases?

The community structures of two large, real world social networks, provided by Telenor Research, were analyzed towards the end. One of which, for the graph representing the cell phone network of Telenors customers, has an accompanying modularity of 0.8996. This value is strikingly high, and there's no doubt that the network admits a community structure. The result obtained on the graph representing the SMS exchanges between customers is interesting mainly because the number of communities in the top level is low, only 110. Although, the value of modularity is also high, 0.8422, it is the low number of communities that allow for easy visualization and clustering, that is interesting here.

The formulation of the Degree-rank method should be altered. Through careful analysis of its output on networks with homogeneous degree distributions, it has become clear that during its first iteration, it may form several small communities that are never dissolved. Perhaps the dispersing phase of Community-dissolve could be put to more use here? After all, the Community-dissolve method, although well motivated, seemingly failed to help the Louvain method unveil partitionings, especially in weighted networks, where it was very sensitive to increasing values of μ_t. However, the shift in focus from sequential to simultaneous moves could be used as a building block for other methods, or as a stepping stone for building a different dispersing phase for the Louvain method.

References

[1] Numpy for matlab users. `http://wiki.scipy.org/NumPy_for_Matlab_Users`. Accessed 23/06/2014.

[2] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[3] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. `http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154`, 2009.

[4] Michele Benzi and Christine Klymko. Total communicability as a centrality measure. *Journal of Complex Networks*, 1(2):124–149, 2013.

[5] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10:8, October 2008.

[6] Geoffrey S Canright and Kenth Engø-Monsen. Introducing network analysis. *Telektronikk. v1*, 2008.

[7] Fan Chung and Linyuan Lu. Connected components in random graphs with given expected degree sequences. *Annals of Combinatorics*, 6(2):125–145, 2002.

[8] Fan Chung, Linyuan Lu, and Van Vu. Spectra of random graphs with given expected degrees. *Proceedings of the National Academy of Sciences*, 100(11):6313–6318, 2003.

[9] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[10] Thomas M Cover and Joy A Thomas. Entropy, relative entropy and mutual information. 1991.

[11] Leon Danon, Albert Diaz-Guilera, Jordi Duch, and Alex Arenas. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(09):P09008, 2005.

[12] Ernesto Estrada and Juan A Rodriguez-Velazquez. Subgraph centrality in complex networks. *Physical Review E*, 71(5):056103, 2005.

[13] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[14] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[15] MathWorks Inc. Matlab, 2013.

[16] John G. Ekerdt James B. Rawlings et al. Gnu/octave. `www.gnu.org/software/octave/`, 2013.

[17] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

[18] Donald Ervin Knuth. *The Stanford GraphBase: a platform for combinatorial computing*, volume 4. Addison-Wesley Reading, 1993.

[19] V. Krebs. `http://www.orgnet.com/`. Accessed 23/06/2014.

[20] Andrea Lancichinetti and Santo Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Physical Review E*, 80(1):016118, 2009.

[21] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: a comparative analysis. *Physical review E*, 80(5):056117, 2009.

[22] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4):046110, 2008.

[23] Elizabeth A Leicht and Mark EJ Newman. Community structure in directed networks. *Physical review letters*, 100(11):118703, 2008.

[24] Ian XY Leung, Pan Hui, Pietro Lio, and Jon Crowcroft. Towards real-time community detection in large networks. *Physical Review E*, 79(6):066107, 2009.

[25] Alex Martelli. *Python in a Nutshell*. O'Reilly, 2009.

[26] Aaron F McDaid, Derek Greene, and Neil Hurley. Normalized mutual information to evaluate overlapping community finding algorithms. *arXiv preprint arXiv:1110.2515*, 2011.

[27] David Mehrle, Amy Strosser, and Anthony Harkin. Walk modularity and community structure in networks. *arXiv preprint arXiv:1401.6733*, 2014.

[28] Marina Meilă. Comparing clusterings by the variation of information. In *Learning theory and kernel machines*, pages 173–187. Springer, 2003.

[29] Marina Meilă. Comparing clusterings—an information based distance. *Journal of Multivariate Analysis*, 98(5):873–895, 2007.

[30] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[31] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Phys. Rev. E*, 74:036104, Sep 2006.

[32] M. E. J. Newman. Spectral methods for network community detection and graph partitioning. *ArXiv e-prints*, July 2013.

[33] M. J. Olsen. Community detection in social networks. Technical report, Norwegian University of Science and Technology, 12 2013.

[34] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[35] Mukund Seshadri, Sridhar Machiraju, Ashwin Sridharan, Jean Bolot, Christos Faloutsos, and Jure Leskove. Mobile call graphs: beyond power-law and log-normal distributions. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 596–604. ACM, 2008.

[36] Josef Stein. `http://en.wikipedia.org/wiki/Binary_GCD_algorithm`. Accessed 23/06/2014.

[37] Lovro Šubelj and Marko Bajec. Unfolding communities in large complex networks: Combining defensive and offensive label propagation for core extraction. *Physical Review E*, 83(3):036103, 2011.

[38] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research*, 9999:2837–2854, 2010.

[39] W. W. Zachary. Zachary's karate club, 1977.

A Technical Appendix

A.1 Compressed Sparse Row Format

Sparse matrix formats are used to store (no surprise) sparse matrices. A sparse matrix is a matrix that has more zero entries than non-zero entries. A sparse matrix format therefore saves only the non-zero entries, and the different formats differ in what way the entries are stored. The compressed sparse row format is a sparse matrix format implemented by three arrays. One array holds the entries of the matrix, and is called `data`. The array `indices` points to the column indices for the corresponding entry in `data`. The final array, `indptr`, is the index pointer of the matrix, specifying what portions of the first two arrays corresponds to what row. This is done such that the column indices for the values in row i are found in `indices[indptr[i]:indptr[i+1]]` and the corresponding values in `data[indptr[i]:indptr[i+1]]`. The `:` is the slice operator giving the values *from and until* the indices in front and behind it.

Another example of a different storage format is the dictionary-of-keys format, which is a dictionary of dictionaries, such that the ij'th entry is found at `dok_matrix[i][j]`. The dictionary-of-keys implementation has a default entry such that if the ij-pair specified is not found in the dictionary, 0 is returned.

A.2 Communities Object

A disjoint community structure could perhaps be represented in memory by a disjoint-set forest data structure. Without going into to much detail, a disjoint-set forest is a data structure that constructs trees out of the sets, such that the root element of the tree is the set label (community label). To find the affiliation of some vertex you must recurse up the tree to the root node. The disjoint-set forest requires only $\mathcal{O}(n)$ memory for n vertices, but it's best implementation yields a time complexity of $\mathcal{O}(\log n)$ as it's worst-case performance when merging communities, determining community affiliation of a vertex and when constructing the sets. However, if all vertices in a tree are connected directly to the root-node, all operations take constant time. The problem with this data structure, is that it can only merge the sets, not dissolve them. Hence, moving a vertex out of it's community c, where $\mid c \mid > 1$ would be impossible. Because of this, the communities object in the python implementation is not implemented as a disjoint-set forest. Instead there are a dictionary with keys the community labels, and values the corresponding set of vertices, as well as a n long list, where the i'th entry determines the community affiliation, c_i of vertex i. This uses twice the amount of memory, $\mathcal{O}(2n)$, but has constant time operations for retrieving a community and determining a vertex' affiliation. There's not much magic going on in the class `Communities` in Listing 8, but the `Communities.move`-method is worth a look. More interesting is the `ModCommunities` found in Listing 10 that extends `Communities`. It's special feature is that it holds a dictionary of the modularities of all communities as well. It was developed for use with the Community-dissolve method, so the modularity dictionary is really

a `heapdict` [9], or a sorted dictionary if you prefer, to allow the extraction of the community with the lowest modularity.

A.3 NumPy and SciPy

SciPy was a clear requirement of this implementation, as a sparse data structure for storing the adjacency matrices was necessary. The first versions of the implementation used NumPy arrays for all lists and data structures, as NumPy is known to be much faster and more memory efficent for big volume numerical calculations. This is because NumPy stores it's arrays in what is basically a `C` array. The obvious difference for a Python user is that a Python list can store any object, while the NumPy array can only store numbers of a prior specified type, e.g. integers or floats. During the development of this thesis we have discovered that the NumPy arrays, although super fast for dot products and matrix operations, are very slow at indexing. As an example, consider the list that holds the degree sequence k. Our (only) use case is looking up elements individually, and although it's stored as a lightweight float object in the NumPy array, the key is that it has to be converted into a Python float object, and this process is time consuming. The memory is really no issue, a Python list with a million integer entries only uses 72 bytes more memory than the NumPy version, so any machine with 4 GB of RAM should be able to run the methods (perhaps not label propagation) on networks of size $\mathcal{O}(10^6)$.

A.4 Structure

In Figure 12 we've made an attempt to outline the structure of the program. The file `main.py` is run from the command line, with the path to the dataset and what method to use as arguments. If the method is the Diffusion and Propagation Algorithm, the file `labelprop.py` will handle everything until it outputs communities using `export_communities.py`. If it the method specified is one of the three other's however, `community_detection.py` has a function that handles the outer loop structure for all methods. If the methods have converged the community structure is outputted.

B Code Listings

[9] `https://github.com/DanielStutzbach/heapdict`

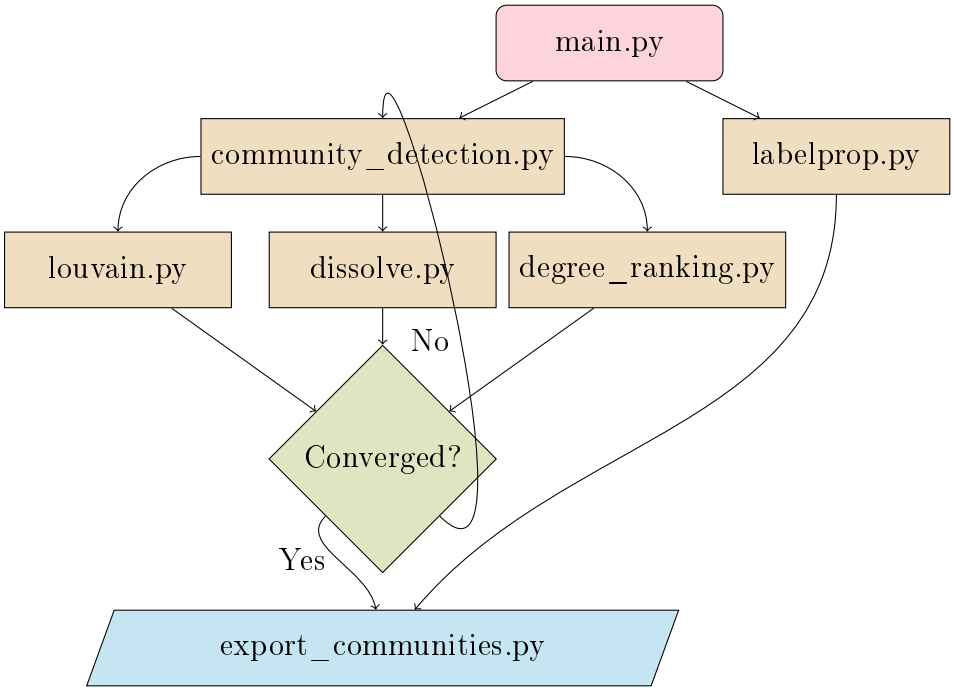


Figure 12: Structure of the community detection code.

Listing 1: main.py

```
import numpy as np
from export_communities import Exporter
from visexport import Viswriter
from csdexport import Csdwriter
import argparse
from scipy import sparse
import os
from community_detection import community_detect
from utils import Method
from utils import Arguments
from utils import Graph

def initialize(A, filepath, args):
    """
    Set up a Graph (named tuple) and instantiate some objects based on
    the arguments passed to the program, then run the algorithm
    specified by args.method.

    Args:
    A: A symmetric SciPy CSR-matrix
    filepath: Path to the file from which A was loaded
    args: All the arguments provided to the program

    """
    filename, ending = os.path.splitext(filepath)

    k = np.array(A.sum(axis=1), dtype=float).reshape(-1,).tolist()
    m = 0.5*A.sum()
    n = A.shape[0]
    G = Graph(A, m, n, k)

    prop = True if args.prop else False
    exporter = Exporter(filename, G.n, prop) if args.output else
       None
    cytowriter = None
    if args.visualize:
        cytowriter = Viswriter(filename, args.vizualize[0],
                               args.vizualize[1], A)

    analyzer = Csdwriter(filename) if args.csd else None
    tsh = args.treshold if args.treshold else 0.02
    verbose = args.verbose if args.verbose else False
    dump = args.dump if args.dump else False

    if args.prop:
        import labelprop
        method = Method.prop
    elif args.rank:
        method =  Method.rank
    elif args.dissolve:
        method = Method.dissolve
    else:
        method = Method.luv
```

```
53
54      arguments = Arguments(exporter, cytowriter, analyzer,
55                            tsh, verbose, dump, method)
56
57      if arguments.verbose:
58          print("File␣loaded.␣{}␣nodes␣in␣the␣network␣and␣total␣
                weight␣"
59                "is␣{}".format(G.n, G.m))
60      if arguments.method == Method.prop:
61          labelprop.propagate(G, arguments)
62      else:
63          community_detect(G, arguments)
64
65  def get_graph(filepath):
66      """
67      Load the matrix saved at filepath.
68
69      Args:
70      filepath: path to file holding a sparse matrix
71
72      Returns:
73      A: SciPy CSR matrix
74
75      """
76      filename, ending = os.path.splitext(filepath)
77      if ending == '.mat':
78          from scipy import io
79          A = sparse.csr_matrix(io.loadmat(filepath)['mat'], dtype=
                float)
80      elif ending == '.csv':
81          A = sparse.csr_matrix(np.genfromtxt(filepath, delimiter=','
                ),
82                                dtype=float)
83      elif ending == '.gml':
84          import networkx as nx
85          A = nx.to_scipy_sparse_matrix(nx.read_gml(filepath), dtype=
                float)
86      elif ending == '.dat':
87          adjlist = np.genfromtxt(filepath)
88
89          if adjlist.shape[1] == 2:
90              data = np.ones(adjlist.shape[0])
91              if np.min(adjlist) == 1:
92                  adjlist -= 1 # 0 indexing
93          else:
94              data = adjlist[:, 2]
95              if np.min(adjlist[:, :-1]) == 1:
96                  adjlist[:, :-1] -= 1 # 0 indexing
97          A = sparse.coo_matrix((data,
98                                 (np.array(adjlist[:,0], dtype=int),
99                                 np.array(adjlist[:,1], dtype=int))),
100                                dtype=float).tocsr()
101
102     elif ending == '.gz' or ending == '.txt':
103         filename = os.path.splitext(filename)[0]
104         import networkx as nx
```

```
105         A = nx.to_scipy_sparse_matrix(
106                 nx.read_weighted_edgelist(filepath, delimiter =' ')
                        ,
107                 dtype=float)
108     else:
109         raise IOError("Could not parse file")
110     return A
111
112 def main():
113     """
114     Parses all arguments passed in the command line and loaded the
            graph
115     located at args.path_to_file. Calls initialize if all arguments
116     are valid
117
118     """
119     parser = argparse.ArgumentParser()
120     parser.add_argument("path_to_file",
121                         help="Specify the path of the data set")
122     parser.add_argument("-t", "--treshold", type=float,
123                         help="Specify an modularity treshold used
                                in the \
124                         first phase. Default is 0.002")
125     parser.add_argument("-v", "--verbose", action="store_true",
126                         help="Turn verbosity on")
127     parser.add_argument("-o", "--output", action="store_true",
128                         help="Output community structure to .txt
                                file"
129                           "in ./results/")
130     parser.add_argument("--dump", action="store_true",
131                         help="Dump communities into pickle file")
132     parser.add_argument("-c", "--csd", action="store_true",
133                         help="Output component sizes")
134     parser.add_argument("-vis", "--visualize", nargs='+', type=int,
135                         help="Export communitiy structure to
                                vizualize with \
136                                     e.g. gephi:\
137                                     arg[0] pass# that should be the
    vertices \
138                                     arg[1] pass# that indicates the
    community \
139                                     structure")
140     parser.add_argument("-p", "--prop", action="store_true",
141                         help="Use labelpropagation algorithm")
142     parser.add_argument("-r", "--rank", action="store_true",
143                         help="Use degree-rank algorithm")
144     parser.add_argument("-d", "--dissolve", action="store_true",
145                         help="Use community-dissolve algorithm")
146
147     args = parser.parse_args()
148
149     if os.path.isfile(args.path_to_file):
150             try:
151                 A = get_graph(args.path_to_file)
152             except IOError:
153                 print("This file extension is not recognized.")
```

```
154                 return
155             initialize(A, args.path_to_file, args)
156     else:
157         print("Please␣provide␣a␣valid␣input␣file")
158 
159 if __name__ == '__main__':
160     main()
```

Listing 2: louvain.py

```
1  from utils import Method
2  from labels import Labels
3  from communities import Communities
4  from modularity_communities import ModCommunities as ModComs
5  from louvain import louvain
6  from degree_ranking import degree_rank
7  from dissolve import community_dissolve
8  import modularity
9  import numpy as np
10 from utils import Graph
11 from scipy import sparse
12 import time
13 
14 def community_detect(G, args):
15     """
16     The outer loop that does all modularity based community
       detection
17     algorithms. Runs the respective methods, and constructs the
18     community network.
19 
20     Args:
21     G: Graph named tuple
22     args: All arguments, including the args.method specifying which
23           method to run.
24 
25     """
26     i = 1
27     t = time.time()
28     old_q = modularity.diagonal_modularity(G.A.diagonal(), G.k, G.m
       )
29 
30     while True:
31         if args.method == Method.luv:
32             C = Communities(xrange(G.n), G.k)
33             q = louvain(G, C, old_q, args.tsh)
34         elif args.method == Method.dissolve:
35             C = ModComs(xrange(G.n), G)
36             q = community_dissolve(G, C, old_q, args.tsh)
37         elif args.method == Method.rank:
38             C = Labels(xrange(G.n), G.k, G.A.diagonal())
39             q = degree_rank(G, C, old_q, args)
40         else:
41             raise Exception("What␣are␣you␣doing␣here.")
42 
43         coms = C.dict_renamed
```

```
44
45          if args.exporter:
46              args.exporter.write_nodelist(coms)
47
48          if not (q > old_q):
49              print("It took {} seconds".format(time.time() - t))
50              if not args.verbose:
51                  print("pass: {}. # of communities: "
52                        "{}. Q = {}".format(i-1, len(coms), q))
53              if args.exporter:
54                  args.exporter.close()
55                  print('Community structure outputted to .txt-file')
56              if args.analyzer:
57                  args.analyzer.show()
58                  print 'CSD dumped to file'
59              return
60
61
62          A = community_network(G.A, coms)
63          k = np.array(A.sum(axis=1), dtype=float).reshape(-1,).
                tolist()
64          m = 0.5*A.sum()
65          n = A.shape[0]
66          G = Graph(A, m, n, k)
67
68          if args.dump:
69              C.dump(i)
70          if args.cytowriter:
71              args.cytowriter.add_pass(coms, G.A)
72          if args.analyzer:
73              args.analyzer.add_pass(coms)
74
75          if args.verbose:
76              print("pass: {}. # of coms: {}. Q = {}".format(i, len(
                    coms), q))
77
78          old_q = q
79          i += 1
80
81  def community_network(A, communities):
82      """
83      The second phase of the Louvain algorithms consists of making
            the
84      community network in which the communities of the first phase
            are
85      nodes in a new network.
86
87      Args:
88      A: Adjacency matrix of the graph
89      coms: A dictionary with keys from 0 to n-1. Values are lists of
90            nodes.
91
92      Returns:
93      B: Adjacency matrix of the community network
94
95      """
```

```
96
97      B = community_affiliation_matrix(communities, A.shape[1])
98      return B.dot(A.dot(B.T))
99
100 def community_affiliation_matrix(coms, n):
101     """
102     Constructing the matrix of community affiliation. Entry ij
            indicates
103     that vertex i is in community j.
104
105     Args:
106     coms: A dictionary with keys from 0 to n-1. Values are lists of
107           nodes.
108     n: The number of vertices in the graph. A.shape[1]
109
110     Returns:
111     A csr matrix indicating community affiliation of vertex i. Each
            row
112     has one 1-entry, and is per node. The columns are communities.
113
114     """
115     # must make sure that dictionary is sorted
116     keys = sorted(coms)
117     ivec = np.array([k for k in keys for j in coms[k]])
118     jvec = np.array([v for r in keys for v in coms[r]])
119     vals = np.ones(len(jvec))
120     coo = sparse.coo_matrix((vals, (ivec, jvec)), shape=(len(keys),
            n))
121     return sparse.csr_matrix(coo)
```

Listing 3: louvain.py

```
1  import modularity as mod
2  import functions as fns
3
4  def louvain(G, C, init_q, tsh):
5      """
6      Find the communities of the graph represented by A using the
           first
7      phase of the Louvain method.
8
9      Args:
10     A: Adjacency matrix in CSR format
11     m: 0.5 * A.sum()
12     n: A.shape[1] the number of vertices in the graph
13     k: degree sequence
14     args: flags and objects for data export etc.
15
16     """
17     n = G.n
18     k = G.k
19     mod_gain = mod.modularity_gain
20     move = C.move
21     new_q = init_q
22
```

```
    while True:
        old_q = new_q

        for i in fns.yield_random_modulo(n):
            (c, movein, moveout) = mod_gain(G, C, i)
            gain = movein + moveout
            if gain > 0:
                move(i, c, k[i])
                new_q += gain

        if new_q - old_q < tsh:
            break

    return new_q
```

Listing 4: dissolve.py

```
import modularity as mod
import functions as fns
# from louvain import louvain

def community_dissolve(G, C, init_q, tsh):
    """
    Run first the first phase of Louvain(with a bit modified
       notation),
    then run dissolve once.

    """
    luv_q = luvxdiss(G, C, init_q, tsh)
    dis_q = dissolve(G, C, luv_q)

    return dis_q

def luvxdiss(G, C, init_q, tsh):
    """
    Find the communities of the graph represented by A using the
       Louvain
    method.

    Args:
    A: Adjacency matrix in CSR format
    m: 0.5 * A.sum()
    n: A.shape[1] the number of vertices in the graph
    k: degree sequence
    args: flags and objects for data export etc.

    """
    n = G.n
    k = G.k
    mod_gain = mod.modularity_gain_new_notation
    move = C.move
    new_q = init_q

    while True:
        old_q = new_q
```

```
37          no_moves = set(C.communities.keys())
38          for i in fns.yield_random_modulo(n):
39              (c, movein, moveout) = mod_gain(G, C, i)
40              gain = movein - moveout + 2*C.node_mods[i]
41              if gain > 0:
42                  no_moves.discard(c)
43                  move(i, c, k[i], movein, moveout, C.node_mods[i])
44                  new_q += gain
45
46          if new_q - old_q < tsh:
47              break
48      return new_q
49
50  def dissolve(G, C, init_q):
51      """
52      Extracts the community with the lowest modularity and tries to
53      'dissolve' it (move all vertices) repeatedly. When all
            communities
54      have been considered it is finished.
55
56      Args:
57      G: Graph named tuple
58      C: Community structure
59      init_q: the initial modularity
60
61      Returns:
62      q: the modularity of the network after a round of dissolve
63
64      """
65      def move(i, dest):
66          """ Move vertex i to dest in our community object """
67          if dest != -1:
68              C.move(i, dest, k[i], movein[i],
69                     moveout[i], quv[dest])
70          else:
71              C.move(i, dest, k[i], movein[i],
72                     moveout[i], C.node_mods[i])
73          quv[dest] = 0.00 # Only add this the first time.
74
75      k = G.k
76      # q = init_q
77      num_dissolved = 0
78      while True:
79          c, (seen, q_c) = C.pop()
80
81          if seen:
82              print("We dissolved: {} communities".format(
                    num_dissolved))
83              return C.network_modularity
84
85          (node2c, c2node, movein,
86           moveout, quv, best) = mod.mass_modularity(G, C, C[c], c)
87
88          if sum(movein.values()) + sum(quv.values()) > q_c:
89              num_dissolved += 1
90              for dest, nodes in c2node.iteritems():
```

```
91                  for i in nodes:
92                      move(i, dest)
93
94      print("We␣dissolved:␣{}␣communities".format(num_dissolved))
95      return C.network_modularity
```

Listing 5: degree_ranking.py

```
1  from modularity import get_gain
2  from utils import rank
3
4  def degree_rank(G, C, q, arguments):
5      """ Finds the communities of A by the degree-rank method. """
6      consider = rank(G.k)
7      not_seen = set(xrange(G.n))
8      while True:
9          new_q, moved = degree_rank_inner(G, C,
10                                          consider, not_seen, q,
                                                arguments)
11         not_seen = set(xrange(G.n))
12         if new_q - q <= arguments.tsh:
13             break
14         q = new_q
15     return new_q
16
17 def degree_rank_inner(G, C, consider, not_seen, old_q, args):
18     """
19     Establish communities around the vertices specified in consider
           .
20
21     Args:
22
23     G: Graph object
24     C: Community structure
25     consider: The vertices we are establish communities around
26     not_seen: Set of nodes marked as not not seen
27     old_q: The modularity of the network when this function is
           called
28     args: Namedtuple of arguments
29
30     Returns:
31     q: modularity of the network
32
33     """
34     A, m, n, k = G
35     q = old_q
36     moved = set([])
37     index = 0
38     i = consider[index]
39
40     while True:
41         not_seen.discard(i)
42         nbs = A.indices[A.indptr[i]:A.indptr[i+1]]
43         for j in nbs:
44             not_seen.discard(j)
```

```
45              if C.nodes[i] != C.nodes[j] and j not in moved:
46                  movein, moveout = get_gain(G, C, j, C.nodes[i])
47                  if movein + moveout > 0:
48                      moved.add(j)
49                      moved.add(i)
50                      C.move(j, C.nodes[i], k[j])
51                      q += movein + moveout
52
53          if not_seen:
54              while True:
55                  index += 1
56                  try:
57                      next = consider[index]
58                  except IndexError:
59                      return q, moved
60
61                  if next not in moved:
62                      i = next
63                      break
64          else:
65              return q, moved
```

Listing 6: labelprop.py

```
1  from __future__ import division
2  from operator import itemgetter
3  from collections import defaultdict
4  import functions as fns
5  import modularity
6  import copy
7  import numpy as np
8  from labels import Labels
9  from utils import Graph
10 from community_detection import community_network
11 MAX_ITER = 100
12
13 def propagate(G, args):
14     """ Start the Diffusion and Propagation Algorithm """
15     G.A.data = np.repeat(1, G.A.data.shape[0]) # convert to
           unweighted
16     k = np.array(G.A.sum(axis=1), dtype=float).reshape(-1,).tolist
           ()
17     G = Graph(G.A, len(G.A.data) / 2, G.n, k)
18     C = dpa(G, args)
19     print("Found␣{}␣communities".format(len(C.dict_renamed)))
20     print("Modularity␣=␣{}".format(modularity.modularity(G, C)))
21     if args.exporter:
22         args.exporter.write_nodelist(C.dict_renamed)
23         args.exporter.close()
24         print('Community␣structure␣outputted␣to␣.txt-file')
25
26 def dalpa(G, C, offensive=False):
27     """
28     Defensive and offensive diffusion and label propagation
           algorithm.
```

```
29
30      Propagates labels along the graph until an equilibrium is
            reached.
31
32      """
33      print("Running␣Dalpa.␣Offensive={}".format(offensive))
34      global MAX_ITER
35      A, m, n, k = G
36      num_iter = 0
37      delta = 0.0
38      while num_iter < MAX_ITER:
39          num_moves = 0
40
41          for i in fns.yield_random_modulo(G.n):
42              indices = A.indices[A.indptr[i]:A.indptr[i+1]]
43              scores = defaultdict(float)
44              for j in indices:
45                  if j != i:
46                      if not offensive:
47                          first_term = C.p[j]
48                      else:
49                          first_term = (1 - C.p[j])
50                      scores[C.nodes[j]] += first_term * (1.0 - delta
                            * C.d[j])
51
52              old = C.nodes[i]
53
54              if scores:
55                  new = max(scores.iteritems(), key=itemgetter(1))[0]
56
57                  if scores[old] < scores[new]:
58                      C.move(i, new, k[i])
59                      dist = n * 10
60                      C.p[i] = 0.0
61                      C.internal[i] = 0
62                      for v in A.indices[A.indptr[i]:A.indptr[i+1]]:
63                          if v != i:
64                              if C.nodes[v] == new:
65                                  if C.d[v] < dist:
66                                      dist = C.d[v]
67                                  C.internal[i] += 1
68                                  C.internal[v] += 1
69                                  if not offensive:
70                                      C.p[i] += C.p[v] / C.internal[v
                                            ]
71                                  else:
72                                      C.p[i] += C.p[v] / k[v]
73
74                              elif C.nodes[v] == old:
75                                  C.internal[v] -= 1
76
77                      C.d[i] = dist + 1
78                      num_moves += 1
79
80          ratio = num_moves / n
81          if ratio < 0.5 and num_iter > 0:
```

```
82              delta = ratio
83          else:
84              delta = 0.0
85
86          num_iter += 1
87          print num_moves
88          if num_moves == 0:
89              break
90
91      if num_iter >= MAX_ITER:
92          print "reached␣max␣iter"
93      print "found␣{}␣communities".format(len(C))
94
95  def dpa(G, args):
96      """
97      Diffusion and propagation algorithm.
98
99      """
100     # Run defensive dalpa
101     defensive_C = Labels(xrange(G.n), G.k, G.A.diagonal())
102     dalpa(G, defensive_C, offensive=False)
103     defensive_Q = modularity.modularity(G, defensive_C)
104
105     # Construct the community network
106     A_C = community_network(G.A, defensive_C.dict_renamed)
107     G_C = Graph(A_C,
108                 G.m,
109                 A_C.shape[1],
110                 np.array(A_C.sum(axis=1), dtype=float).reshape(-1,)
                        .tolist()
111                 )
112
113     # Run offensive dalpa on the community network
114     offensive_C = Labels(xrange(G_C.n), G_C.k, G_C.A.diagonal())
115     dalpa(G_C, offensive_C, offensive=True)
116     offensive_Q = modularity.modularity(G_C, offensive_C)
117
118     # if the modularity of the offensive run is higher than the
            modularity
119     # of the defensive run, we wish to transfer the labels/
            communities of
120     # community network onto the real network.
121
122     if offensive_Q > defensive_Q and len(offensive_C) > 1:
123         clustering = [-1] * G.n
124         # community in G_C holds communities in G
125         for high_level_c, low_level_cs in offensive_C:
126             for low_level_c in low_level_cs:
127                 for node in defensive_C[low_level_c]:
128                     clustering[node] = high_level_c
129
130         # Keep in mind that the internal edges are wrong below
131         defensive_C = Labels(clustering, G.k, G.A.diagonal())
132         defensive_Q = offensive_Q
133
134         # # If we need to fix the intra-community edges:
```

```
135         # defensive_C.internal = [0] * G.n
136         # for c, nodes in defensive_C:
137         #     for u in nodes:
138         #         for v in G.A.indices[G.A.indptr[u]:G.A.indptr[u
                +1]]:
139         #             if v != u:
140         #                 if defensive_C.nodes[v] == c:
141         #                     defensive_C.internal[u] += 1
142 
143     if len(offensive_C) == 1:
144         defensive_C = Labels(xrange(G.n), G.k, G.A.diagonal())
145         defensive_C = bdpa(G, defensive_C)
146         return defensive_C
147     else:
148         print "RECURSING"
149         # print defensive_C.dict
150         largest = list(defensive_C[defensive_C.largest])
151         largest.sort()
152 
153         # mapping vertex in subset-graph to vertex in real graph
154         mapping = {i: j for i, j in enumerate(largest)}
155 
156         A_subset = G.A[largest, :][:, largest]
157         G_subset = Graph(
158             A_subset,
159             A_subset.sum() / 2,
160             A_subset.shape[1],
161             np.array(A_subset.sum(axis=1), dtype=float).reshape(-1,
                    ).tolist()
162             )
163         recursive_C = dpa(G_subset, args)
164         new_C = copy.deepcopy(defensive_C)
165 
166         for c, nodes in recursive_C:
167             new_C.insert_community([mapping[node] for node in nodes
                    ], G.k)
168 
169         new_Q = modularity.modularity(G, new_C)
170 
171         if new_Q > defensive_Q:
172             defensive_C = new_C
173 
174         return new_C
175 
176 def bdpa(G, C):
177     print "Running␣BDPA"
178     dalpa(G, C, offensive=False)
179     defensive_Q = modularity.modularity(G, C)
180     print "defensive_Q␣=␣{}␣".format(defensive_Q)
181     new_C = copy.deepcopy(C)
182     for c, nodes in C:
183         median = np.median([C.p[j] for j in nodes])
184         for i in C[c]:
185             if new_C.p[i] <= median:
186                 new_C.move(i, -1, G.k[i])
187                 new_C.d[i] = 0
```

```
188                 new_C.p[i] = 0
189                 new_C.internal[i] = G.A[i,i]
190     # Fix internal edges
191     new_C.internal = [0] * G.n
192     for i in xrange(G.n):
193         for j in G.A.indices[G.A.indptr[i]:G.A.indptr[i+1]]:
194             if new_C.nodes[i] == new_C.nodes[j]:
195                 new_C.internal[i] += 1
196     dalpa(G, new_C, True)
197     offensive_Q = modularity.modularity(G, new_C)
198     print "offensive_Q = {} ".format(offensive_Q)
199     if offensive_Q > defensive_Q:
200         return new_C
201     else:
202         return C
```

Listing 7: modularity.py

```
1  from collections import defaultdict
2  import numpy as np
3  import numexpr as nr
4
5  def diagonal_modularity(diag, k, m):
6      """
7      Calculates the modularity when all vertices are in their own
8      community.
9
10     Args:
11     diag: numpy array of length n holding the diagonal entries of
           some
12             matrix
13     k: degree sequence of the above mentioned matrix. n long.
14     m: The total weight of the graph. 0.5 * A.sum()
15
16     """
17     ks = np.array(k)
18     return (1.0/(2*m))*nr.evaluate("sum(diag)") -(1/(4*m**2))*nr.
           evaluate("sum(ks**2)")
19
20 def modularity(G, C):
21     """
22     Calculates the global modularity by summing over each community
           .
23     Should be deprecated.
24
25     Args:
26     G: Graph named tuple
27     C: Community structure
28
29     Returns:
30     q: Modularity of the network with the provided community
           structure
31
32     """
33     A, m, n, k = G
```

```
34      q = 0.0
35      for com, c in C:
36          rowslice = A[c,:]
37          data = rowslice.data
38          indices = rowslice.indices
39          q += ((1.0/(2*m))*np.sum(data[np.in1d(indices, c)]) -
40               (C.strength[com]/(2*m))**2)
41      return q
42
43  def single_node_modularity(G, i):
44      """
45      Calculates the modularity of an isolated node.
46      Args:
47
48      A: Adjacency matrix of the graph
49      k: Degree sequence of the graph
50      m: The total weight of the graph. 0.5 * A.sum()
51      i: the vertex considered
52
53      Returns:
54      A float representing the modularity of the isolated node.
55
56      """
57      return G.A[i,i]/(2*G.m) - (G.k[i]/(2*G.m))**2
58
59  def modularity_of_partition(A, k, m, nodes):
60      """
61      Calculates the modularity of the group consisting of the
           vertices in
62      'nodes'.
63
64      Args:
65
66      A: Adjacency matrix of the graph
67      k: Degree sequence of the graph
68      m: The total weight of the graph. 0.5 * A.sum()
69      nodes: A list of vertices
70
71      Returns:
72      q: The modularity of the partition defined by 'nodes'
73
74      """
75      rowslice = A[nodes,:]
76      data = rowslice.data
77      indices = rowslice.indices
78      q = ((1.0/(2*m))*np.sum(data[np.in1d(indices, nodes)]) -
79           (sum(k[i] for i in nodes)/(2*m))**2)
80      return q
81
82  def modularity_gain(G, C, i):
83      """"
84      Calculates the modularity gain of moving vertex i into the
85      community of its neighbors. NB: Follows the notation of Olsen
           (2013)
86
87      Args:
```

```
    G: Graph named tuple
    C: Community structure
    i: A vertex whose neighbors we iterate over.

    Returns:
    Destination, modularity gain and modularity loss of the move.

    """
    A, m, n, k = G
    indices = A.indices[A.indptr[i]:A.indptr[i+1]]
    data = A.data[A.indptr[i]:A.indptr[i+1]]

    movein = {}
    k_i = k[i]
    c_i = C.nodes[i]
    const = k_i/(2.0*m**2)
    moveout = (2.0/(4.0*m**2))*k_i*(C.strength[c_i] - k_i)
    max_movein = (-1, -1.0)
    for ind,j in enumerate(indices):
        aij = data[ind]
        c_j = C.nodes[j]
        if c_j == c_i:
            if i != j:
                moveout -= aij/m
            continue

        if c_j in movein:
            movein[c_j] += aij/m
        else:
            movein[c_j] = aij/m - const*C.strength[c_j]

        if movein[c_j] > max_movein[1]:
            max_movein = (c_j, movein[c_j])

    if not movein:
        return (-1, -1.0, 0.0)

    return (max_movein[0], max_movein[1],  moveout)

def modularity_gain_new_notation(G, C, i):
    """
    The new notation essentially means that moveout now includes
       q_i.

    Args:
    G: Graph named tuple
    C: Community structure
    i: A vertex whose neighbors we iterate over.

    Returns:
    Destination, modularity gain of Destination and modularity loss
    the old community of i.

    """
    A, m, n, k = G
    indices = A.indices[A.indptr[i]:A.indptr[i+1]]
```

```
143     data = A.data[A.indptr[i]:A.indptr[i+1]]
144
145     movein = {}
146     k_i = k[i]
147     c_i = C.nodes[i]
148     movein = {}
149     moveout = -2*k_i*C.strength[c_i]/((2*m)**2)
150     max_movein = (-1, -1.0)
151     for ind, j in enumerate(indices):
152         aij = data[ind]
153         c_j = C.nodes[j]
154         if c_j == c_i:
155             moveout += aij / m
156             continue
157         try:
158             movein[c_j] += aij/m
159         except KeyError:
160             movein[c_j] = aij/m - 2*k_i*C.strength[c_j]/((2*m)**2)
161
162         if movein[c_j] > max_movein[1]:
163             max_movein = (c_j, movein[c_j])
164
165     if not movein:
166         return (-1, -100.0, 0.0)
167     return (max_movein[0], max_movein[1],  moveout)
168
169 def get_gain(G, C, i, dest):
170     """
171     Calculates and returns the gain of moving i to dest.
172
173     Args:
174     i: the integer label of the vertex to be moved
175     dest: the label of the proposed community
176     C: the community object
177
178     Returns:
179     Two floats, movein and moveout, such that the modularity after
            the
180     move is q += movein + moveout.
181
182     """
183     A, m, n, k = G
184     data = A.data[A.indptr[i]:A.indptr[i+1]]
185     indices = A.indices[A.indptr[i]:A.indptr[i+1]]
186     k_i = k[i]
187     c_i = C.nodes[i]
188     movein = - k_i*C.strength[dest]/(2.0*m**2)
189     moveout = (2.0/(4.0*m**2))*k_i*(C.strength[c_i] - k_i)
190     for ind,j in enumerate(indices):
191         aij = data[ind]
192         c_j = C.nodes[j]
193         if c_j == c_i:
194             if i != j:
195                 moveout -= aij/m
196             continue
197         elif c_j == dest:
```

```
198             movein += aij/m
199
200     return movein, moveout
201
202 def mass_modularity(G, C, nodes, c):
203     """
204     Calculates the modularity gain of moving each of the nodes
205     to the best match.
206
207     Args:
208     G: Graph object
209     nodes: list of nodes in community
210     C: Community structure
211     c: original affiliation of nodes
212
213     Returns:
214     node2c: dict holding best match for vertex i
215     c2node: dict holding the vertices going to community c
216     dqins: holds the global gain of moving vertex i to node2c[i].
217     dqouts: holds the global loss of  --"--
218     quv: the modularity of the subsets that is moved. If only one
           vertex
219           is moved to a community, this is q_i.
220     best_move: the (node, community) move that has the highest
221                modularity gain associated to it
222
223     """
224     A, m, n, k = G
225
226     node2c = {}
227     c2node = defaultdict(set)
228     dqins = {}
229     dqouts = {}
230     quv = defaultdict(float)
231     best_move = (-1, -1)
232
233     for i in nodes:
234         indices = A.indices[A.indptr[i]:A.indptr[i+1]]
235         data = A.data[A.indptr[i]:A.indptr[i+1]]
236         nbs = set([])
237         crossterms = defaultdict(float)
238         movein = {}
239         k_i = k[i]
240         moveout = -2*k_i*C.strength[c]/((2*m)**2)
241         max_movein = (-1, 0.0)
242
243         for ind, j in enumerate(indices):
244             aij = data[ind]
245             k_j = k[j]
246             c_j = C.nodes[j]
247             if c_j == c:
248                 moveout += aij/m
249
250                 try:
251                     nc_j = node2c[j]
252                 except KeyError:
```

```
                    continue
                else:
                    if nc_j != -1:
                        qij = aij/(2*m) - (k_i*k_j)/(2*m)**2
                        nbs.add(j)
                        crossterms[nc_j] += 2*qij
                continue

            try:
                movein[c_j] += aij/m
            except KeyError:
                movein[c_j] = aij/m - 2*k_i*C.strength[c_j]/(2*m)
                    **2

            if movein[c_j] > max_movein[1]:
                max_movein = (c_j, movein[c_j])

        dest, q_in = max_movein
        node2c[i] = dest
        c2node[dest].add(i)
        dqins[i] = q_in
        dqouts[i] = moveout

        if q_in - moveout > best_move[1]:
            best_move = (i, q_in - moveout)

        qi = C.node_mods[i]
        quv[dest] += qi
        quv[dest] += crossterms[dest]

        for node in c2node[dest] - (nbs | set([i])):
            qij = -2*k[i]*k[node]/(2*m)**2
            quv[dest] += qij

    return node2c, c2node, dqins, dqouts, quv, best_move[0]
```

Listing 8: communities.py

```

class Communities(object):
    """
    This class represents a community structure. A collection of
    disjoint sets(communities) such that all vertices in a graph
       are found in
    exactly one such set.

    """
    def __init__(self, iterable, k):
        """
        Initialize the community-object by a iterable specifying a
        vertex -> community mapping.

        Args:
        iterable: Iterable such that the i'th element specifies
                  the community affiliation of vertex i.
```

```
17          k: Degree sequence of same length as 'iterable'.
18
19          """
20          self.nodes = list(iterable)
21          self.communities = {}
22          self.strength = {}
23          self.used = set([])
24          for i, c in enumerate(iterable):
25              if c not in self.communities:
26                  self.communities[c] = set([i])
27                  self.strength[c] = k[i]
28                  self.used.add(c)
29              else:
30                  self.communities[c].add(i)
31                  self.strength[c] += k[i]
32
33
34      def move(self, i, s, k_i):
35          """
36          Move the vertex i to community s.
37
38          Args:
39          i: the integer label of the vertex that is moving
40          s: the destination(new community) of i
41          k_i: the degree of i
42
43          """
44
45          s_i = self.nodes[i]
46
47          # remove i from it's community
48          self.communities[s_i].remove(i)
49
50          # if there's no nodes left, remove community from dicts
51          if not self.communities[s_i]:
52              del self.communities[s_i]
53              del self.strength[s_i]
54          # key might not be in strength
55          try:
56              self.strength[s_i] -= k_i
57          except KeyError:
58              pass
59
60          if s == -1:
61              # Isolate vertex i
62              j = self._unused_key()
63              self.communities[j] = set([i])
64              self.strength[j] = k_i
65              self.nodes[i] = j
66          else:
67              self.nodes[i] = s
68              self.strength[s] += k_i
69              self.communities[s].add(i)
70
71      def insert_community(self, nodes, k):
72          newkey = self._unused_key()
```

```
73          self.communities[newkey] = set([])
74          self.strength[newkey] = 0
75          for node in nodes:
76              self.move(node, newkey, k[node])
77
78      def delete_community(self, c, k):
79          nodes = self.communities[c].copy()
80          for node in nodes:
81              self.move(node, -1, k[node])
82
83      def _unused_key(self):
84          for j in xrange(4*len(self.nodes), 0, -1):
85              if j not in self.used:
86                  self.used.add(j)
87                  return j
88          raise Exception("Couldn't find key")
89
90      def neighbors(self, x):
91          a = self.communities[self.get_community(x)]
92          try:
93              b = a.copy()
94              b.remove(x)
95              return list(b)
96          except TypeError:
97              return []
98
99      def size(self, c):
100         return len(self.communities[c])
101
102     @property
103     def dict(self):
104         return {key: list(value) for key, value in
105                 self.communities.iteritems()}
106
107     @property
108     def dict_renamed(self):
109         # sort keys
110         keys = sorted(self.communities.keys())
111         # rename communities and return
112         return {i:list(self.communities[x]) for i, x in enumerate(
                keys)}
113
114     def dump(self, i):
115         import cPickle as pickle
116         pickle.dump(self, open("".join(['pickled_', \
117             'coms', str(i), '.p']), "wb"))
118
119     def recluster(self, com_dict, k):
120         for name, coms in com_dict.iteritems():
121             for c_i in coms[1:]:
122                  for i in self.getnodes(c_i):
123                     self.move(i, coms[0], k[i])
124     @property
125     def largest(self):
126         largest = (-1, -1)
127         for c, nodes in self.communities.iteritems():
```

```
128             if len(nodes) > largest[1]:
129                 largest = (c, len(nodes))
130         return largest[0]
131
132     def __iter__(self):
133         for key in self.communities.keys():
134             yield (key, list(self.communities[key]))
135
136     def __getitem__(self, c_i):
137         try:
138             com = self.communities[c_i]
139         except KeyError:
140             com = set([])
141         return com
142
143     def __len__(self):
144         return len(self.communities)
```

Listing 9: labels.py

```
1 from communities import Communities
2
3 class Labels(Communities):
4     """ Extension of Community structure to handle label
          propagation """
5     def __init__(self, iterable, k, diagonal):
6         super(Labels, self).__init__(iterable, k)
7         self.internal = [diagonal[i] for i in iterable]
8         self.d = [0.0] * len(self.nodes)
9         self.p = [1.0/len(self.nodes)] * len(self.nodes)
```

Listing 10: modularity_communities.py

```
1  import modularity
2  from labels import Labels
3  from heapdict import heapdict
4
5  class ModCommunities(Labels):
6
7      def __init__(self, iterable, G):
8          """
9          Modularity holds {key: (0/1, priority)} pairs
10         """
11         super(ModCommunities, self).__init__(iterable, G.k, G.A.
               diagonal())
12         self.modularity = heapdict()
13         self.node_mods = {}
14         self.changed = False
15         self.network_modularity = 0.0
16
17         for i in iterable:
18             q = modularity.single_node_modularity(G, i)
19             self.modularity[i] = (0, q)
20             self.node_mods[i] = q
21             self.network_modularity += q
```

```
22
23      def pop(self, i=0):
24          """
25          Pop the community with the lowest modularity, push it back
26          (but with the first value of the tuple 1 and not 0) and
                return
27          the item.
28
29          Args:
30          i: the index to pop. Default 0.
31
32          Returns:
33          (x, y, z): x the key of the community, y=0/y=1, z
                modularity of
34                     the community.
35
36          """
37          item_key, (item_seen, item_val) = self.modularity.peekitem
                ()
38          self.modularity[item_key] = (1, item_val)
39          return item_key, (item_seen, item_val)
40
41      def move(self, i, s, k_i, movein, moveout, quv):
42          """
43          Move a vertex from it's community to the community s.
44
45          Args:
46          i: the integer label of the vertex to be moved
47          s: the destination of vertex i. May be -1 to indicate that
                we
48             want to isolate the vertex.
49          k_i: k[i], the degree of vertex i
50          movein: The global modularity gain of moving vertex i to s.
51          moveout: The global modularity loss of moving vertex i from
                it's
52                   community.
53          quv: q_s + movein + quv = q_s* the new modularity. If
54               mass_modularity is being used, remember to only add
                     this
55               quantity once.
56
57          """
58          s_i = self.nodes[i]
59
60          # remove i from it's community
61          self.communities[s_i].remove(i)
62
63          # if there's no nodes left, remove community from dicts
64          if not self.communities[s_i]:
65              del self.communities[s_i]
66              del self.strength[s_i]
67              self.network_modularity -= self.modularity[s_i][1]
68              del self.modularity[s_i]
69
70          # key might not be in strength, since we might have deleted
                it
```

```
        try:
            self.strength[s_i] -= k_i
        except KeyError:
            #The community has been deleted.
            pass

        # same goes for modularity
        try:
            (seen, mod) = self.modularity[s_i]
        except KeyError:
            #The community has been deleted.
            pass
        else:
            self.modularity[s_i] = (seen, mod - moveout + quv)
            self.network_modularity -= moveout
            self.network_modularity += quv

        if s == -1:
            # Isolate vertex i
            j = self._unused_key()
            self.communities[j] = set([i])
            self.strength[j] = k_i
            self.nodes[i] = j
            self.modularity[j] = (0, quv)
            self.network_modularity += quv

        else:
            self.nodes[i] = s
            self.communities[s].add(i)
            self.strength[s] += k_i
            (seen, mod) = self.modularity[s]
            self.modularity[s] = (seen, mod + movein + quv)
            self.network_modularity += (movein + quv)

    def unsee_all(self):
        """
        Sets the first entry in the value tuple to 0 for all
           entries
        in the modularity-heapdict.

        """
        for key, (seen, val) in self.modularity.iteritems():
            self.modularity[key] = (0, val)
```

Listing 11: transform.py

```
import argparse
import os
import numpy as np
from scipy import io, sparse
from scipy.sparse import linalg
import main

def matrix_power(mtx, exp):
    """ Return the exp power of (mtx + I) """
```

```
10      I = sparse.identity(mtx.shape[1], dtype=float, format='csr')
11      A = mtx + I
12      for i in xrange(int(exp)-1):
13          A = A.dot(mtx + I)
14      return A
15
16  def walk_generator(A):
17      """ Return the walk-generating function of A. """
18      I = sparse.identity(A.shape[1], dtype=float)
19      inv_mat = linalg.inv((I-A).tocsc()).tocsr()
20      return inv_mat
21
22  def exponentiate(mat):
23      """ Return the matrix exponential of A, exp(A). """
24      exp_mat = linalg.expm(mat.tocsc()).tocsr()
25      return exp_mat
26
27  def reciprocal_ties(mat):
28      """ Symmetrize A considering reciprocal ties. """
29      A = mat.todok()
30      B = sparse.dok_matrix(A.shape)
31      for (i, j), aij in A.iteritems():
32          if (j,i) in A:
33              val = aij + A[j, i]
34              B[i, j] = val
35              B[j, i] = val
36
37      return B.tocsr()
38
39  def symmetrize(mat):
40      """ Symmetrize by taking the mean of the aij and aji entries
            """
41      return (mat + mat.T)/2
42
43  def extract_largest_component(mat):
44      """ Extract the largest component using scipy's method """
45      num_comp, affiliation = sparse.csgraph.connected_components(mat
            )
46      if num_comp == 1:
47          return mat
48      max_comp = np.argmax(np.bincount(affiliation))
49      indices = np.arange(mat.shape[0])
50      indices = indices[affiliation == max_comp]
51
52      return mat[indices, :][:, indices]
53
54  def edge_restriction(restrictee, restrictor):
55      """
56      Restrict the edges of restrictee by the edges of restrictor:
57
58      Returns:
59      A matrix with the elements of restrictee where restrictors
            elements
60      are nonzero.
61
62      """
```

```
63      indptr = restrictor.indptr
64      indices = restrictor.indices
65      nz = restrictor.nonzero()
66      data = np.array(restrictee[nz[0], nz[1]], dtype=float)[0]
67      return sparse.csr_matrix((data, indices, indptr), dtype=float)
68
69
70  def power_main():
71      parser = argparse.ArgumentParser()
72      parser.add_argument("path_to_input",
73                          help="Specify␣the␣path␣of␣the␣input␣data␣
                               set")
74      parser.add_argument("-p", "--power", type=int,
75                          help="Specify␣to␣which␣power␣to␣raise␣the␣
                               matrix␣to")
76      parser.add_argument("-w", "--walk", help="Calculate␣(I-A)^-1",
77                          action="store_true")
78      parser.add_argument("-e", "--exp", help="Calculate␣exp(A)",
79                          action="store_true")
80      parser.add_argument("-r", "--restrict",
81                          help="Restrict␣the␣elements␣of␣the␣"
82                          "transformed␣matrix␣to␣the␣coordinates␣of␣
                               the␣nonzero␣"
83                          "elements␣of␣the␣original␣matrix.",
84                          action="store_true")
85      parser.add_argument("--recip", action="store_true",
86                          help="Symmetrize␣by␣reciprocal␣ties")
87      parser.add_argument("--symmetrize", action="store_true",
88                          help="Symmetrize␣by␣the␣mean␣of␣entry␣ij␣
                               and␣ji")
89      parser.add_argument("-lcc", "--components", action="store_true"
            ,
90                          help="Extract␣the␣largest␣connected␣
                               component")
91      parser.add_argument("path_to_output", \
92          help="Specify␣where␣to␣save␣output")
93
94      args = parser.parse_args()
95      in_path = args.path_to_input
96      out_path = args.path_to_output
97      if os.path.isfile(in_path):
98          filename, ending = os.path.splitext(in_path)
99          out_path, out_ending = os.path.splitext(out_path)
100         try:
101             A = main.get_graph(in_path)
102         except IOError:
103             print("File␣format␣not␣recognized")
104         else:
105             if args.power:
106                 mat = matrix_power(A, args.power)
107             elif args.walk:
108                 mat = walk_generator(A)
109             elif args.exp:
110                 mat = exponentiate(A)
111             else:
112                 mat = A
```

```
113
114             if args.recip:
115                 mat = reciprocal_ties(mat)
116             elif args.symmetrize:
117                 mat = symmetrize(mat)
118
119             if args.components:
120                 mat = extract_largest_component(mat)
121
122             if args.restrict:
123                 mat = edge_restriction(mat, A)
124
125             if out_path:
126                 io.savemat(out_path, {'mat': mat}, do_compression=
                        True,
127                            oned_as='row')
128     else:
129         print("Specify a valid input-file")
130
131 if __name__ ==  '__main__':
132     power_main()
```

Listing 12: functions.py

```
1  import random
2  """
3  Functions for generating the cyclic group [0,...n-1]. Use instead
       of
4  random.
5
6  Functions:
7      yield_random_modulo(n) <- generate the numbers in 0,...n-1
8      bin_gcd(a, b) <- calculate the gcd of a and b fast
9
10 """
11
12 def yield_random_modulo(n):
13     """
14     Generates the cyclic group 0 through n-1 using a number
15     which is relative prime to n.
16
17     """
18     while True:
19         rand = random.random()
20         rand = int(rand * n) # number between 0 and n
21         if bin_gcd(rand, n) == 1:
22             break
23     i = 1
24     while i <= n:
25         yield i*rand % n
26         i += 1
27
28 def bin_gcd(a, b):
29     """
30     Return the greatest common divisor of a and b using the binary
```

```
    gcd algorithm.

    """
    if a == b or b == 0:
        return a
    if a == 0:
        return b

    if not a & 1:
        if not b & 1:
            return bin_gcd(a >> 1, b >> 1) << 1
        else: # b is odd
            return bin_gcd(a >> 1, b)
    if not b & 1:
        return bin_gcd(a, b >> 1)
    if a > b:
        return bin_gcd((a - b) >> 1, b)

    return bin_gcd((b - a) >> 1, a)
```

Listing 13: utils.py

```
"""
This module defines an Enum and some namedtuples for use throughout
the whole lib.

"""

from collections import namedtuple
from enum import Enum
from operator import itemgetter

Method = Enum('Method', 'luv␣rank␣dissolve␣prop')

Arguments = namedtuple('Arguments',
    ['exporter',
    'cytowriter',
    'analyzer',
    'tsh',
    'verbose',
    'dump',
    'method']
    )

Graph = namedtuple('Graph', ['A', 'm', 'n', 'k'])

def rank(sequence):
    """
    Return the index from the original sequence the element
    has in the sorted array.

    """
    ranked = zip(*sorted(enumerate(sequence), key=itemgetter(1))
        [::-1])[0]
    return list(ranked)
```

Listing 14: graphing.py

```
import numpy as np
import argparse
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import pandas as pd
import os
try:
    import seaborn as sns
    cols = np.array(sns.color_palette("husl", 8))
    paired = np.array(sns.color_palette("Paired", 10)[2:])

except ImportError:
    print 'Fancy plots disabled as Seaborn is not installed'

"""

This module should load the results from a tab-delimitered csv-file
into some data-structure, and then plot it

"""

def import_and_format(filename, separator):
    """
    Read csv file and infer from the filename the parameters
    of the network.

    Will fail if the csv is not structured as expected.

    """
    data = pd.read_csv(filename, sep=separator)
    data['n'] = np.nan
    data['mu_t'] = np.nan
    data['mu_w'] = -1
    data['transformation'] = 'A'

    for i in xrange(data.shape[0]):
        filename = data.loc[i, 'File']
        properties = filename.split('_')
        properties[-1] = properties[-1][:-4]
        for p in properties:
            if p.startswith('no'):
                data['transformation'][i] = str(p)
            elif p.startswith('n'):
                data['n'][i] = int(p[1:])
            elif p.startswith('mut'):
                data['mu_t'][i] = int(p[3:])
            elif p.startswith('muw'):
                data['mu_w'][i] = int(p[3:])
            else:
                data['transformation'][i] = str(p)
    return data

def plot(filepath):
```

```
55      """
56      Reads the output of suite.py and
57      plots the performance of the methods.
58
59      """
60      data = import_and_format(filepath, '\t')
61      print np.unique(data.mu_t)
62      data.mu_t = data.mu_t / 100
63      data.mu_t[data.mu_t == 0.35] = 0.035
64      data.mu_t[data.mu_t == 0.65] = 0.065
65      data.mu_t[(data.mu_t == 0.2) & (data.mu_w != -1)] = 0.02
66
67      data.mu_w = data.mu_w.astype(float)
68      data.mu_w = data.mu_w / 100
69      data.transformation[data.transformation == "pow2"] = "A^2"
70      data.transformation[data.transformation == "pow3"] = "A^3"
71      data.transformation[data.transformation == "exp"] = "exp(A)"
72      data.method[data.method == "luv"] = "Louvain"
73      data.method[data.method == "rank"] = "Rank"
74      data.method[data.method == "prop"] = "Propagation"
75      data.method[data.method == "dissolve"] = "Dissolve"
76      means = data.groupby(['method',
77                            'n',
78                            'mu_t',
79                            'mu_w',
80                            'transformation']).mean()
81
82      methods = ["Louvain", "Rank", "Dissolve"]
83      transformations = ["A", "A^2", "A^3", "exp(A)"]
84
85      binary = means.query("mu_w == -0.01")
86      binary.index = binary.index.droplevel("mu_w")
87
88      coolness = means.query("mu_t == 0.99")
89
90      weighted = means.query("mu_t < 0.10")
91      weighted = weighted.drop("norest", level="transformation")
92
93
94
95
96      for metric in ["NMI", "NVI"]:
97          # Compare methods on binary networks
98
99          for i, n in enumerate((1000, 5000)):
100             fig = plt.figure(figsize = (4, 4), tight_layout=True)
101             axis = fig.add_subplot(1,1,1)
102
103             xs = binary.xs([n, "A"], level=['n', 'transformation'])
104             xs[metric].unstack().T.plot(ax=axis, color=cols[[0, 2,
                    5, 6]])
105             axis.set_ylim(0.0, 1.0)
106             axis.set_xlim(0.2, 0.9)
107             # axis.set_title("n = {}".format(n))
108
109             if metric == "NVI":
```

```
110                 axis.set_ylabel("Normalized␣Variation␣of␣
                        Information")
111                 axis.legend(loc="upper␣left")
112             else:
113                 axis.set_ylabel("Normalized␣Mutual␣Information")
114                 axis.legend(loc="lower␣left")
115             if i > 0:
116                 axis.legend().set_visible(False)
117
118             fig.savefig("../Master/figures/binary_compare/
                    binary_compare_{}_{}_xx.png".format(metric, n), dpi
                    =600)
119             plt.close(fig)
120         continue
121         # Plot all transformations of binary networks for all
                methods
122         for n in (1000, 5000):
123             for i, method in enumerate(methods):
124                 fig = plt.figure(figsize=(4, 4), tight_layout=True)
125                 axis = fig.add_subplot(1,1,1)
126                 xs = binary.xs([n, method], level=['n', 'method'])
127                 xs[metric].unstack().plot(ax=axis, color=cols[[3,
                        1, 5, 7]])
128                 axis.set_ylim(0.0, 1.0)
129                 axis.set_xlim(0.2, 0.9)
130                 if metric == "NVI":
131                     axis.set_ylabel("Normalized␣Variation␣of␣
                            Information")
132                     axis.legend(loc="upper␣left")
133                 else:
134                     axis.set_ylabel("Normalized␣Mutual␣Information"
                            )
135                     axis.legend(loc="lower␣left")
136                 if i > 0:
137                     axis.legend().set_visible(False)
138                 fig.savefig("../Master/figures/binary_methods/
                        binary_methods_n{}_{}_{}".format(n, metric,
                        method), dpi=600)
139                 plt.close(fig)
140
141         # Compare methods on weighted graph
142
143         for i, n in enumerate((1000, 5000)):
144             for j, mu in enumerate((0.05, 0.08)):
145                 fig = plt.figure(figsize = (4, 4), tight_layout=
                        True)
146                 axis = fig.add_subplot(1,1,1)
147                 xs = weighted.xs([n, mu, "A"], level=['n', 'mu_t',
                        'transformation'])
148                 xs[metric].unstack().T.drop("Propagation", 1).plot(
                        ax=axis, color=cols[[0,2,6]])
149                 axis.set_ylim(0.0, 1.0)
150                 axis.set_xlim(0.2, 0.9)
151                 if metric == "NVI":
152                     axis.set_ylabel("Normalized␣Variation␣of␣
                            Information")
```

```
153                     axis.legend(loc="upper␣left")
154                 else:
155                     axis.set_ylabel("Normalized␣Mutual␣Information"
                            )
156                     axis.legend(loc="lower␣left")
157                 # axis.set_title("n = {}, mu_t = {}".format(n, mu
                        *10))
158 
159                 if j != 0:
160                     axis.legend().set_visible(False)
161                 fig.savefig("../Master/figures/weighted_compare/
                        weighted_compare_{}_n{}_mu{}.png".format(metric,
                         n, int(mu*100)), dpi=600)
162                 plt.close(fig)
163 
164         # Plot the transformations on the weighted network
165         for n in ((5000,)):
166             for mu in [0.02, 0.035, 0.05, 0.065, 0.08]:
167                 for i, method in enumerate(["Louvain"]):
168                     fig = plt.figure(figsize = (4, 4), tight_layout
                            =True)
169                     axis = fig.add_subplot(1,1,1)
170                     print np.unique(data.mu_t)
171                     print n, mu, method, metric
172                     xs = weighted.xs([method, n, mu], level=['
                            method', 'n', 'mu_t'])
173 
174                     xs_col = xs[metric].unstack()
175                     print xs_col
176                     xs_col.plot(ax=axis, color=cols[[3, 1, 5, 7]])
177 
178                     axis.set_ylim(0.0, 1.0)
179                     axis.set_xlim(0.2, 0.9)
180                     if metric == "NVI":
181                         axis.set_ylabel("Normalized␣Variation␣of␣
                                Information")
182                         axis.legend(loc="upper␣left")
183                     else:
184                         axis.set_ylabel("Normalized␣Mutual␣
                                Information")
185                         axis.legend(loc="lower␣left")
186                     # axis.set_title(method)
187                     if i > 0:
188                         axis.legend().set_visible(False)
189                     fig.savefig("../Master/figures/weighted_methods
                            /weighted_methods_n{}_mu{}_{}_{}.png".format
                            (n, int(mu*100), metric, method), dpi=600)
190                     plt.close(fig)
191 
192         # Plot the results for transformation when mu_t = mu_w
193         for i, method in enumerate(methods):
194             fig = plt.figure(figsize = (4, 4), tight_layout=True)
195             axis = fig.add_subplot(1,1,1)
196             xs = coolness.xs([5000, 0.99, method], level=['n', "
                    mu_t", 'method'])
197             xs_col = xs[metric].unstack()
```

```
198             xs_col.plot(ax=axis, color=cols[[3, 1, 5, 7]])
199             axis.set_ylim(0.0, 1.0)
200             axis.set_xlim(0.2, 0.9)
201             if metric == "NVI":
202                 axis.set_ylabel("Normalized␣Variation␣of␣
                        Information")
203                 axis.legend(loc="upper␣left")
204             else:
205                 axis.set_ylabel("Normalized␣Mutual␣Information")
206                 axis.legend(loc="lower␣left")
207             # axis.set_title(method)
208             if i > 0:
209                 axis.legend().set_visible(False)
210             fig.savefig("../Master/figures/mutmuw/mutmuw_{}_{}.png"
                    .format(metric, method), dpi=600)
211             plt.close(fig)
212
213         # Compare methods
214         fig = plt.figure(figsize = (4, 4), tight_layout=True)
215         axis = fig.add_subplot(1,1,1)
216         xs = coolness.xs([5000, 0.99, 'A'], level=['n', "mu_t", '
                transformation'])
217         xs_col = xs[metric].unstack().T
218         xs_col.plot(ax=axis, color=cols[[0, 2, 5, 6]])
219         axis.set_ylim(0.0, 1.0)
220         axis.set_xlim(0.2, 0.9)
221         if metric == "NVI":
222             axis.set_ylabel("Normalized␣Variation␣of␣Information")
223             axis.legend(loc="upper␣left")
224         else:
225             axis.set_ylabel("Normalized␣Mutual␣Information")
226             axis.legend(loc="lower␣left")
227
228         fig.savefig("../Master/figures/mutmuw/mutmuw_compare_{}.png
                ".format(metric), dpi=600)
229         plt.close(fig)
230
231
232
233 if __name__ == '__main__':
234     parser = argparse.ArgumentParser()
235     parser.add_argument("filepath")
236     args = parser.parse_args()
237     if args.filepath and os.path.isfile(args.filepath):
238         plot(args.filepath)
239     else:
240         print("Provide␣a␣valid␣file")
```

Listing 15: suite.py

```
1 from __future__ import division
2 import argparse
3 import labelprop
4 import main
5 import numpy as np
```

```
6  import os
7  import tester
8  from multiprocessing import Pool
9  from export_communities import Exporter
10 from utils import Graph, Arguments, Method
11 from community_detection import community_detect
12
13 def get_right_column(comlist, ncom):
14     """
15     Return the column of comlist that has closest to ncom unique
           entries
16
17     """
18     smallest = (-1, 99999999999)
19     for i in xrange(comlist.shape[1]):
20         uniq = len(np.unique(comlist[:,i]))
21         if abs(uniq - ncom) < smallest[1]:
22             smallest = (i, uniq - ncom)
23     return comlist[:, smallest[0]]
24
25 def get_best_column(result_matrix):
26     """ Get the index of the column that minimizes the NVI. """
27     indices = result_matrix[:, -2].argmin(axis=0)
28     try:
29         idx = indices[0]
30     except IndexError:
31         idx = indices
32     return idx
33
34 def get_files(dir):
35     """
36     Recursively walks through all folder within 'dir' and outputs
37     the paths of all the files together with the file specifying
38     the ground truth community structure.
39
40     """
41     file_list = []
42     for root, dirs, files in os.walk(dir):
43         dir_files = []
44         for f in files:
45             if f.endswith('truth.dat'):
46                 truth = os.path.join(root, f)
47             elif (not f.endswith('walk.mat') and
48                   not f.endswith('.DS_Store')):
49                 dir_files.append(os.path.join(root, f))
50         if files:
51             file_list.append((dir_files, truth))
52     return file_list
53
54
55 def format(f, mi, nmi, vi, nvi, n_found, n_known, method):
56     """ Return the arguments as a tab-delimitered string """
57     return "{}\t{}\t{}\t{}\t{}\t{}\t{}\t{}\n".format(
58                     f, mi, nmi, vi, nvi, n_found, n_known, method)
59
60 class Run(object):
```

```
61      """
62      This class is used to run one of the community detection
           methods,
63      gather the community structure, and test it against the ground
64      truth.
65
66      """
67      def __init__(self, truth, method):
68          """
69          Instantiate the object with a ground truth and a method.
70
71          Args:
72          truth: the ground truth community structure of the network
73                 it will be run on.
74          method: A constant indicating what method the object will
               run
75                   when called.
76
77          """
78          self.truth = truth
79          self.method = method
80
81      def __call__(self, f):
82          """
83          When the object is called like a function, we run the
               method
84          specified by self.method on the dataset in file 'f'.
85
86          Args:
87          f: path to file where the network corresponding the the
               ground
88             truth community structure lies.
89
90          Returns:
91          A string of test results.
92
93          """
94          print(self.method)
95          print(f)
96          G = initialize_graph(f)
97          known = tester.parse(self.truth)
98          known -= 1
99          exporter = Exporter(f, G.n, False)
100         arguments = Arguments(exporter, None, None, 0.02,
101                               False, False, self.method)
102         if self.method == Method.prop:
103             labelprop.propagate(G, arguments)
104             found = arguments.exporter.comlist[:, -1]
105             numcoms = len(np.unique(found))
106             test_results = tester.test(found, known)
107         else:
108             community_detect(G, arguments)
109             hierarchy = arguments.exporter.comlist[:, 1:] # Exclude
                    the 0...n col
110             colresult = np.empty(shape=(hierarchy.shape[1], 4))
111             lengths = []
```

```
112
113             for j, column in enumerate(hierarchy.T):
114                 lengths.append(len(np.unique(column)))
115                 colresult[j, :] = tester.test(column, known)
116
117             idx = get_best_column(colresult)
118             test_results = colresult[idx, :]
119             numcoms = lengths[idx]
120
121         return format(os.path.basename(f), test_results[0],
                test_results[1],
122                 test_results[2], test_results[3], numcoms,
123                 len(np.unique(known)), str(arguments.method).split(
                    '.')[-1])
124
125 def initialize_graph(f):
126     """ Helper method to load file and make the Graph named tuple
            """
127     A = main.get_graph(f)
128     G = Graph(A,
129               0.5*A.sum(),
130               A.shape[1],
131               np.array(A.sum(axis=1), dtype=float).reshape(-1,).
                    tolist()
132               )
133     return G
134
135 def output_to_file(filename, results):
136     """
137     Write the array of result-strings 'results' to the file '
            filename'.
138     Will write a header if one is missing.
139     """
140     with open(filename, 'a+') as output:
141         output.seek(0)
142         if not output.readline():
143             output.write("File\tMI\tNMI\tVI\tNVI\tn_found\tn_known\
                    tmethod\n")
144         else:
145             output.seek(0, 2) # Put cursor at the end of the file.
146         for line in results:
147             output.write(line)
148
149 if __name__ == '__main__':
150     """ Run the tests using the multiprocessing module """
151     parser = argparse.ArgumentParser()
152     parser.add_argument("path_to_dir",
153                         help="Specify␣the␣path␣of␣the␣data␣set")
154     parser.add_argument("n",
155                         help="The␣number␣of␣runs␣for␣each␣data␣set"
                                )
156     args = parser.parse_args()
157
158     result_strings = []
159     def res_app(res):
160         result_strings.append(res)
```

```
161
162     if not os.path.isdir(args.path_to_dir):
163         print("That's␣not␣a␣folder.")
164     else:
165         pool = Pool()
166         files = get_files(args.path_to_dir)
167         for fs, ground_truth in files:
168             for f in fs:
169                 # rank is deterministic
170                 pool.apply_async(Run(ground_truth, Method.rank),
171                     args=(f, ),
172                     callback=res_app)
173                 for i in xrange(int(args.n)):
174                     #Louvain, dissolve and labelprop are not, so we
                            average
175                     pool.apply_async(Run(ground_truth, Method.luv),
176                         args=(f, ),
177                         callback=res_app)
178                     pool.apply_async(Run(ground_truth, Method.
                        dissolve),
179                         args=(f, ),
180                         callback=res_app)
181                     pool.apply_async(Run(ground_truth, Method.prop)
                        ,
182                         args=(f, ),
183                         callback=res_app)
184         pool.close()
185         pool.join()
186         output_to_file('results/results.txt', result_strings)
187         print("Tests␣ended␣just␣fine.")
```

Listing 16: tester.py

```
1
2 """
3 This module containes measures from information theory to compare
4 to clusterings.
5
6 """
7
8 import argparse
9 import numpy as np
10 from scipy import sparse
11 from math import log
12
13 def parse(path):
14     return np.loadtxt(path, dtype=int)[:, -1]
15
16 def log2(x):
17     return log(x, 2)
18
19 def mutual_information(N):
20     hxy = joint_entropy(N)
21     h_known = entropy(N.sum(axis=1)) # row sums
22     h_found = entropy(N.sum(axis=0)) # col sums
```

```
23      return h_known + h_found - hxy
24
25  def joint_entropy(N):
26      H = 0
27      for (i,j) in zip(*N.nonzero()):
28          nij = N[i,j]
29          H += -1 * nij * log2(nij)
30      return H
31
32  def entropy(n):
33      H = 0
34      for e in n:
35          if e !=0:
36              H += -1 * e * log2(e)
37      return H
38
39  def variation_of_information(N):
40      hxy = joint_entropy(N)
41      ixy = mutual_information(N)
42      return hxy - ixy
43
44  def normalized_variation_of_information(N):
45      hxy = joint_entropy(N)
46      ixy = mutual_information(N)
47      return 1 - (ixy/hxy)
48
49  def normalized_mutual_information(N):
50      ixy = mutual_information(N)
51      h_known = entropy(N.sum(axis=1))
52      h_found = entropy(N.sum(axis=0))
53      return 2*ixy/(h_known + h_found)
54
55  def max_mutual_information(N):
56      ixy = mutual_information(N)
57      h_known = entropy(N.sum(axis=1))
58      h_found = entropy(N.sum(axis=0))
59      return ixy/max((h_known,h_found))
60
61  def joint_density(found, known):
62
63      n_found = len(np.unique(found))
64      n_known = len(np.unique(known))
65      # print("{} x {} density".format(n_found, n_known))
66
67      # coo-matrix will sum duplicate entries
68      confusion = np.asarray(
69          sparse.coo_matrix(
70              (np.ones(known.shape[0], dtype=float), (known, found)),
71              shape=(n_known, n_found)
72          ).todense()
73      )
74      return confusion/confusion.sum(dtype=float)
75
76  def test(found, known):
77      N = joint_density(found, known)
78      return (mutual_information(N),
```

```
79              max_mutual_information(N),
80              variation_of_information(N),
81              normalized_variation_of_information(N))
82
83  def main():
84      parser = argparse.ArgumentParser()
85      parser.add_argument("found")
86      parser.add_argument("known")
87      parser.add_argument("--ext", action="store_true",
88                          help="Put␣this␣if␣nodes␣are␣number␣from␣1."
                                )
89      args = parser.parse_args()
90
91      if not (args.found and args.known):
92          print("Please␣specify␣both␣files")
93          return
94
95      found = parse(args.found)
96      known = parse(args.known)
97
98      if args.ext:
99          known -= 1
100     N = joint_density(found, known)
101
102     print("---Testing␣{}␣vs.␣{}---".format(args.found, args.known))
103     print("Variation␣of␣information␣(VI):␣{}".format(
            variation_of_information(N)))
104     print("Normalized␣VI:␣{}".format(
            normalized_variation_of_information(N)))
105     print("Mutual␣Information␣(MI):␣{}".format(mutual_information(N
            )))
106     print("Normalized␣MI:␣{}".format(normalized_mutual_information(
            N)))
107     print("Max-normalized␣MI:␣{}␣\n".format(max_mutual_information(
            N)))
108
109 if __name__ == '__main__':
110     main()
```