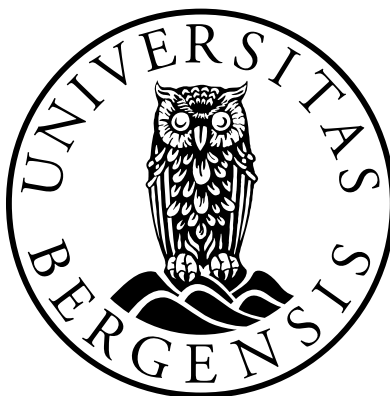# University of Bergen

Department of Informatics

Community Detection in Social Networks

Author:
Erlend Eindride Fasmer

Supervisor:
Jan Arne Telle

April 2015

Abstract

Social networks usually display a hierarchy of communities and it is the task of community detection algorithms to detect these communities and preferably also their hierarchical relationships. One common class of such hierarchical algorithms are the agglomerative algorithms. These algorithms start with one community per vertex in the network and keep agglomerating vertices together to form increasingly larger communities. Another common class of hierarchical algorithms are the divisive algorithms. These algorithms start with a single community comprising all the vertices of the network and then split the network into several connected components that are viewed as communities.

We start this thesis by giving an introductory overview of the field of community detection in part I, including complex networks, the basic groups of community definitions, the modularity function, and a description of common community detection techniques, including agglomerative and divisive algorithms.

Then we proceed, in part II, with community detection algorithms that have been implemented and tested, with refined use of data structures, as part of this thesis. We start by describing, implementing and testing against benchmark graphs the greedy hierarchical agglomerative community detection algorithm proposed by Aaron Clauset, M. E. J. Newman, and Cristopher Moore in 2004 in the article *Finding community structure in very large networks* [5]. We continue with describing and implementing the hierarchical divisive algorithm proposed by Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi in 2004 in the article *Defining and identifying communities in networks* [28]. Instead of testing this algorithm against benchmark graphs we present a community detection web service that runs the algorithm by Radicchi et al. on the collaboration networks in the DBLP database of scientific publications and co-authorships in the area of computer science. We allow the user to freely set the many parameters that we have defined for this algorithm. The final judgment on the results is measured by the modularity value or can be left to the knowledgeable user. A rough description of the design of the algorithms and of the web service is given, and all code is available at GitHub [10] [9].

Lastly, a few improvements both to the algorithm by Radicchi et al. and to the web service are presented.

Acknowledgement

First and foremost, I would like to thank my supervisor, Jan Arne, for continuous encouragement during the last year, for many hours spent on invaluable help and suggestions both with respect to the present text, with the implementation of the algorithms, and with the web service.

I would also like to thank Fredrik Manne for giving a qualified evaluation of the results obtained by running the RECC algorithm on his collaboration network.

I would like to honor my parents, Ole Bernt and Gunn, for a good childhood, for giving me the love of knowledge, and for encouraging me to take higher education.

The Algorithms Group as a whole also deserves acknowledgement for providing a stimulating environment.

Contents

Part I

Preliminaries

Chapter 1

Introduction

1.1 What is community detection?

There are many kinds of networks in the real world and in this thesis we focus on what is known as social networks, that involve some type of social interactions between humans. One such network is the *collaboration network* of all researchers and their coauthor relationships. In this network each researcher is represented by a vertex and two vertices are connected by an edge if the researchers they represent have at least one publication in common. The vertices of real world networks tend to clump together into communities such that there are considerably more edges inside the communities than between them. The aim of the field of *community detection* is to reveal the potential communities of a network and possibly also their hierarchical structure.

Given a well defined social network, community detection has several important applications. For instance, finding the communities of a collaboration network can help discover groups of researchers with expertise in the same area. Another example is online social networking services such as Facebook. Take John, a fictive user of Facebook, as an example. John's network consists of every acquaintance with a Facebook account that he has registered as a friend. John has three clearly distinct communities. The first community consists of his near family, the second community consists of colleagues from his office, and the third community consists of friends from former years in the army. Now consider how Facebook repeatedly suggests that you may know a friend of a friend and that you should possibly add this friend of a friend to your friend list. Should an army friend of John be suggested to add John's grandmother as a friend? Community detection can help in areas such as this one.

1.2 Outline of this thesis

The outline of this thesis is as follows. In part I, preliminaries, we give all the necessary background material that is needed in the main part.

The preliminaries starts out with an introduction to complex networks theory in chapter 2. The chapter gives the reader a soft introduction to the field with some real world examples of complex networks and then proceeds with four important mathematical properties of complex networks. The first property is the scale-free property, meaning that the degree distribution of complex networks often follows a power-law. Then there is the small-world property, meaning that the mean shortest path length, taken over all vertex pairs, is small relative to the total number of vertices in the network. The third property is the community structure, which is the main property of this thesis, meaning that the vertices of complex networks tend to clump together in tightly connected groups. The last property is that one can define meaningful clustering coefficients that measure the level of clustering in the network, either at a global or local level.

In chapter 3 we first present basic community notation, as it is used in the field of community detection. Then we proceed with explaining four basic categories of community definitions, and giving examples of each of them. Even though there are several graph theoretic concepts in this chapter, the graph theoretician may be unfamiliar with some of the notation as the field is dominated by physicists with their own notation.

In chapter 4 we present the modularity function, which is the most well known function that measures the quality of partitions of vertex sets into communities. The maximum obtainable value of the function is discussed, as well as the simplicity of the function and the resulting limitations on the quality of the partitions found with the help of the function. Lastly, an alternative function that resembles the modularity function, called motif modularity, is presented. Both these functions are made use of in the main part.

In the last chapter in the preliminaries, in chapter 5, we discuss some basic algorithms for community detection. We first present graph partitioning and explain its poor ability to be used for community detection. Then we proceed with introducing the field of hierarchical clustering algorithms, the main field of algorithms in this thesis. Hierarchical clustering algorithms can be divided into agglomerative and divisive algorithms. Agglomerative algorithms are bottom-up algorithms that start out with one community per vertex and keeps agglomerating vertices together to form increasingly larger communities. Divisive algorithms, on the other hand, are top-down algorithms that start out with one community comprising all the vertices of the networks and then keeps splitting the initial community into several communities. The last part of the chapter presents several examples of greedy hierarchical algorithms that maximize the modularity function.

In part II of this thesis we discuss and implement two hierarchical algorithms, and incorporate one of them into a community detection web service.

In chapter 7, we describe and implement a greedy hierarchical agglomerative clustering algorithm by Aaron Clauset, Mark E. J. Newman, and Cristopher Moore [5]. This algorithm greedily maximizes the modularity function. The chapter is concluded with some tests on benchmark graphs, and these tests are not altogether successful when we look at the final numbers only. This is partly

due to the simplicity of the modularity function, explained in section 4.3, and partly due to the fact that measuring partitions into communities is not an easy task and there is often not a final and definitive answer. A better approach is to use some sort of scale, e.g. from 1 to 10, to say how good or bad the partition is.

In chapter 8 we describe and implement a divisive clustering algorithm by Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, Domenico Parisi [28]. This algorithm uses a clustering coefficient called the edge clustering coefficient (ECC), that is inspired by the motif modularity function, presented in section 4.4. The ECC is used to split one initial community comprising all the vertices into several smaller communities.

The algorithm by Radicchi et al. is not tested on benchmark graphs. Rather, in chapter 9, in response to a challenge at the end of the article by Radicchi et al., noting that one should interpret the results of a partition of a vertex set into communities on the given network in order to say a final world on the quality of the partition, we present a web service offering community detection in the DBLP database using the algorithm by Radicchi et al. and thus offering everybody a chance to evaluate the results. The algorithm of Radicchi et al. only works on unweighted networks but later the same year Castellano et al. [4] extended the algorithm to also work on weighted networks and we have extended both our algorithm and the web service in line with their paper in order to offer community detection on weighted collaboration networks in the DBLP database. In the weighted case, the number of joint publications that two reserachers have in common is taken into account.

The web service presented in chapter 9 is implemented using the Spring Framework and utilizes the XML API of the DBLP server in order to build the collaboration networks.

In chapter 10 we conclude our thesis and present some suggestions on future research on the algorithm by Radicchi et al. and some improvements of the community detection web service.

All the standard graph theory that is necessary to keep up the pace with this thesis is presented in chapter A in the appendix.

Table 1.1 gives an overview over the algorithms presented in this thesis, as well as where in the thesis they are mentioned.

Section	Year	Short name	Authors	Type	Parent	I
5.1, 5.3.3	1970	KL	Kernighan-Lin [16]	GP		
5.2.2, 8.1.1, 8.1.2	2002	GN	Girvan and Newman [13] [24]	DH		
5.2.2, 8	2004	REB	Radicchi, Castellano, Cecconi, Loreto, and Parisi [28]	DH	GN	
5.2.2, 8, 9	2004	RECC	Radicchi, Castellano, Cecconi, Loreto, and Parisi [28]	DH	GN	X
5.3.1, 5.3.2, 5.3.3	2004	N	Newman [23]	GAH		
5.3.2, 5.3.3, 7, 9	2004	CNM	Clauset, Newman, and Moore [5]	GAH	N	X
5.3.3	2007	WT	Wakita and Tsurumi [35]	GAH	CNM	
5.3.3	2008	SC	Schuetz and Caflisc [30]	GAH	CNM, KL	

Table 1.1: The algorithms studied in this thesis. The column "Section" gives an overview of the sections and chapters of this thesis where the given algorithms are mentioned. The following abbreviations are used in the column "Type", which explains what type of algorithm the given algorithm is. GP = graph partitioning, DH = divisive hierarchical, and GAH = greedy agglomerative hierarchical. The "Parent" column contains the algorithm(s) that the given algorithm builds upon. The column "I" shows which algorithms are implemented in this thesis.

Chapter 2

Complex networks

In recent times, and in particular since the birth of the Internet, it has become common to analyze interactions in the real world by looking at the networks, or graphs, underlying these interactions. These networks can be defined in various ways, and also the terminology used for them is varied.

This chapter on complex networks is based on the book *Complex Networks: structure, robustness, and function* by Reuven Cohen (associate professor in the Department of Mathematics at Bar-Ilan University) and Shlomo Havlin (professor in the Department of Physics at Bar-Ilan University) [6], unless otherwise noted. Cohen and Havlin are two prominent authorities in the field of complex networks. In this thesis, our terminology sticks to that of Reuven and Havlin.

We will first give an introduction to complex networks as found in the real world and then we will proceed with some mathematical properties of these networks.

2.1   Complex networks in the real world

This section presents some examples of real-world complex networks, including computer networks, social networks, and biological networks.

Computer networks consist of computers (vertices) connected by (physical) connections (edges) such as cable or satellite. The sizes of these networks vary, ranging from local area networks (LANs) to wide area networks (WANs). Most computer networks are connected to the Internet, a network of many autonomous systems (ASs), that is, networks run by their respective owners. Data packets are sent between networks with the help of routers. The Internet is either studied at the router level or at the AS level when considered as a complex network. In 2009, the AS network consisted of approximately 10^4 vertices (ASs).

There exist many types of technological networks, including phone networks and transportation networks.

Virtual technological networks are networks where the edges and sometimes even the vertices are logical rather than physical. The most notable example of

such a network is the World Wide Web (WWW), a directed network of HTML pages (vertices), where there is an edge from one page to another if the former links to the latter. Another example is the email network where every individual is represented by a vertex and a directed edge (u, v) exists if v is in the electronic address book of u. Yet another type of network is the phone call graph, which is usually created by phone network operators. In this graph, every phone number is represented by a vertex and a directed edge (u, v) exists if phone number u made a call to v during a certain period, e.g. a day.

Another class of complex networks is social networks, which are the focus of this thesis. These networks encode social interactions between individuals, such as friendships, acquaintances, job relations, etc. One example of such networks is the actor network, where every actor is represented by a vertex and two vertices are connected by an edge if the corresponding actors have played together in some movie. Another example is the scientific coauthorship network where every scientist is represented by a vertex and two vertices are connected by an edge if they have published some paper together. Yet another example is the citation network where every scientist is represented by a vertex and a directed edge (u, v) exists if u cites a paper by v in any of her publications.

A class of well studied complex networks is the biological networks. Several of these networks are logical, e.g. networks representing interactions between proteins or genes, and networks modelling interactions between molecules in the cell's metabolic pathways. Although the actual interaction is physical, the edges represent possible interactions and are therefore deemed logical. Another type of biological network is the physical biological network. Important examples include the nervous system, the neurons in the brain, and the network of blood vessels in an organism.

2.2 Mathematical properties of complex networks

A *complex network* is a graph with structural features that are not present in simple networks such as lattices or random graphs. These structural features are neither purely regular nor purely random and include a vertex degree distribution following something that looks like a power law, a high clustering coefficient, community structure and hierarchical structure. These features and others are described in the following. We start with two properties of complex networks that have received much attention lately: the scale-free and power law property and the small-world property.

Scale-free networks

A network is said to be *scale-free* if its degree distribution follows a power law, at least asymptotically as the number of vertices grows. The degree of a vertex is the number of edges adjacent to it. The degree distribution $P(k)$ of a network is the fraction of vertices in the network with degree k, i.e. $P(k) = \frac{n_k}{n}$, where n is the total number of vertices in the network and n_k is the number of vertices

with degree k. If $P(k)$ follows a power law, then it is approximately equal to $k^{-\gamma}$ for large values of k, where the parameter γ typically is in the range $(2,3)$. The World Wide Web and email networks are examples of real world networks with degree distributions following power laws. A degree distribution resembling a power law has a large number of vertices of low degree and a small number of vertices with high degree, sometimes referred to as *hubs*. The corresponding function plotting the degree distribution looks something like the one in Figure 2.1.
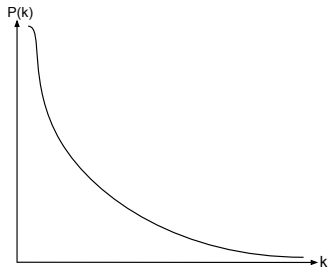
There exists several models for the evolution of a scale-free network, the two most important being the *Barabási-Albert model* (also called the preferential attachment model), and the *configuration model* (also called the Bollobás construction and the Molloy-Reed construction).

The *Barabási-Albert model* is an algorithm that generates random scale-free networks with the preferential attachment property. This means that the more neighbors a vertex has, the higher probability it has of being linked to new vertices introduced to the network. The network is built as follows. We start with a network on n_0 vertices and repeatedly add new vertices to the network. Every time we add a new vertex to the network we connect it to $n \leq n_0$ existing vertices. The probability p_i that the vertex is connected to vertex i is proportional to the number of neighbors that i already has. Formally p_i is defined as

$$p_i = \frac{k_i}{\sum_j k_j} \tag{2.1}$$

Figure 2.1: This is how a graph following a power law looks like, more or less.

where the sum is taken over all the vertices j that exist at the time we add the new vertex.

Small-world networks

A network is said to be a *small-world network* if it exhibits the small-world property, i.e. that the mean shortest path length ℓ, taken over all vertex pairs, is small relative to the total number of vertices, n, in the network. ℓ must not grow faster than logarithmically as the number of vertices tends to infinity, that is, $\ell \to \mathcal{O}(\log n)$ as $n \to \infty$. Note that this definition does not allow one to determine whether an individual network is a small-world network since a collection of graphs are needed in order to rigorously define the property. However, given a specific network, it is possible to compute all the usual graph properties. For instance it is possible make several randomized versions of the

network by randomly rewiring its edges, compute the geodesic distance for each version, and finally compute the mean geodesic distance [27]. The World Wide Web and the metabolic network are examples of small-world networks.

Community structure

Social networks (among other complex networks) usually display *community structure* (which is sometimes referred to as clustering). A network is said to display such structure if the vertices of the network can be partitioned into either overlapping or disjoint sets of vertices such that the number of internal edges exceeds the number of external edges by some reasonable amount. An internal edge is an edge connecting two vertices belonging to the same community whereas an external edge is an edge connecting two vertices of different communities.

Networks displaying a community structure may often display a *hierarchical* community structure as well. This means that the network may be divided into a small amount of large communities and each of these may be divided into several smaller communities. In section 5.2 (in the preliminaries) we will look closer at algorithms that reveal the hierarchical community structure of graphs, and in the main part, in chapter 7 and chapter 8, we will implement and test two such algorithms.

Clustering coefficients

A *clustering coefficient* is a measure of how much the vertices of a network tend to cluster together. Especially in social networks vertices tend to form densely connected groups (communities). There are two different and well known clustering coefficients, a global and a local.

The *global clustering coefficient* C is due to Luce and Perry [20] and is based on the concept of a triple of vertices. A connected triple is an ordered triple (in the set theoretic sense of the word) (a, b, c) of three vertices a, b, and c such that a is connected to b and b to c. The coefficient is defined as

$$C = \frac{3 \times \text{number of triangles}}{\text{number of connected triples}} \qquad (2.2)$$

The factor of 3 is due to the fact that every triangle gives rise to three connected triples, see Figure 2.2 [27].

The *local clustering coefficient* was introduced by Duncan J. Watts and Steven Strogatz in 1998 as a measure used to determine if a network is a small-world network. Each vertex v has its own clustering coefficient C_v, which is defined as the number of edges between the vertices in its neighborhood over the total number of possible edges between these vertices. For an undirected graph the coefficient is then

$$C_v = \frac{2 \cdot |\{\{u, w\} : u, w \in N(v), \{u, w\} \in E\}|}{k_v(k_v - 1)} \qquad (2.3)$$
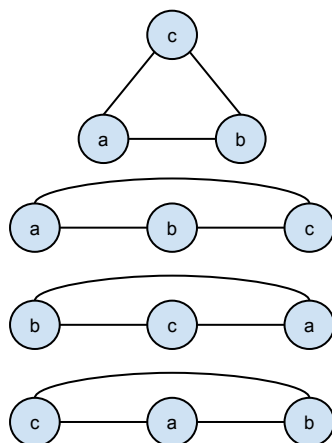
Figure 2.2: A triangle graph consisting of three vertices a, b, and c shown on the top. Below it we have arranged the vertices of the graph in three different ways in order to reveal the three corresponding triples (a, b, c), (b, c, a), and (c, a, b).

where $k_v = |N(v)|$.

In chapter 8 we will introduce a clustering coefficient called the edge clustering coefficient, which is the central concept in the main algorithm of this thesis.

2.3 Random Network Models

In this section we briefly introduce random graph theory and a random graph model called the ER model.

Random graph theory

Random graph theory is the study of a probability space Ω of graphs. Each graph in Ω has a probability attached to it. A certain property π exists with probability p if the total probability of a graph in Ω possessing π is p, or in other words, if the number of graphs in Ω that have property π divided by the total number of graphs in Ω is p.

The Erdős-Rényi Models

The Erdős-Rényi model (henceforth called the ER model) may refer to one of two closely related random graph models. The $G_{n,p}$ model was introduced by Paul Erdős and Alfréd Rényi in 1959. The model generates a graph with n vertices and puts an edge between every pair of vertices with probability p. The $G_{n,M}$ model was introduced independently and contemporaneously by Edgar

Gilbert. In this model a graph is chosen uniformly at random from the space of all graphs with n vertices and M edges. The two models are similar if $M = \binom{N}{2}p$.

In random graph theory the average degree of a graph plays an important role. The degree of vertex i will be denoted k_i and the average degree of a graph will be denoted $\langle k \rangle$. A graph on n vertices with $\langle k \rangle = \mathcal{O}(n^0) = \mathcal{O}(1)$ is called a sparse graph. The *Dictionary of Algorithms and Data Structures* defines a sparse graph as a graph "in which the number of edges is much less than the possible number of edges." [26]

An important characteristic of $G_{n,p}$ is that many of its properties have a related threshold function, $p_t(n)$. When $n \to \infty$ such a property exists with probability 0 if $p < p_t$ and with probability 1 if $p > p_t$, where p_t is the threshold probability. The existence of a giant component, i.e. a large set of connected vertices, is an example of such a property. Erdős and Rényi showed that such a component exists if $\langle k \rangle > 1$ whereas only small components exists when $\langle k \rangle < 1$ and then the size of the largest component is proportional to $\ln n$. When $\langle k \rangle = 1$ a component with size proportional to $n^{2/3}$ appears.

Chapter 3

Communities

There is no universally accepted definition of a community. However, intuitively a community is conceived of as a subset of vertices of a graph with more internal than external edges. Many different community definitions have been introduced. They can roughly be divided into local and global community definitions. The most notable of these definitions will be presented in the following but first we will present a set of basic definitions that often occur in community detection literature.

Note that the rest of this thesis does not depend on section 3.4. This section stands as it does merely for the sake of completeness.

3.1 Basic definitions

In this section we present only the basic definitions from community detection theory that we actually will use later on in this thesis.

In the following definitions $C = (V', E')$ is a subgraph of a graph $G = (V, E)$, where $|V'| = n_c$ and $|V| = n$.

Whereas the degree of a vertex v usually is denoted $d(v)$ in graph theory, it is usually denoted k_v in complex network theory, and we will stick to the latter notation in the rest of this thesis.

Definition 3.1.1. *The **internal degree** $k_v^{int(C)}$ of a vertex v in a community C is the number of edges connecting v to other vertices of C.*

Definition 3.1.2. *The **external degree** $k_v^{ext(C)}$ of a vertex v in a community C is the number of edges connecting v to the rest of the graph.*

A vertex $v \in C$ where $k_v^{ext(C)} = 0$ has thus found a good community in C. If on the other hand $k_v^{int(C)} = 0$, v is disjoint from C and should be assigned to another community.

Definition 3.1.3. *The **internal degree** k_{int}^C of a community C is the sum of the internal degrees of its vertices.*

15

Definition 3.1.4. *The **external degree** k_{ext}^C of a community C is the sum of the external degrees of its vertices.*

Definition 3.1.5. *The **total degree** k^C of a community C is the sum of the degrees of its vertices.*
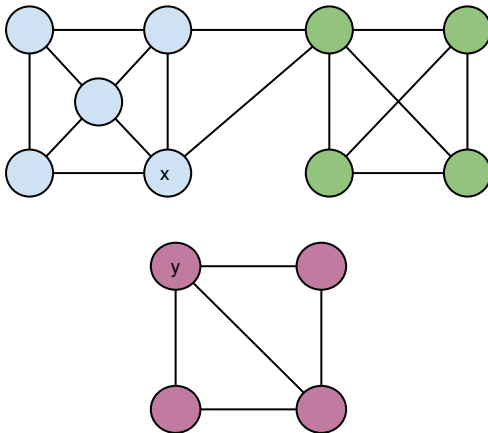
Figure 3.1: A graph consisting of three different communities, B, G, and P, shown in blue, green and purple respectively. $k_x^{int(B)} = 3$, $k_x^{ext(B)} = 2$, $k_{int}^B = 16$, $k_{ext}^B = 2$, $k^B = 18$. Note that $k_y^{ext(P)} = 0$ and thus P is a good community for vertex y.

See Figure 3.1 for an example where the definitions above is used.

3.2 Local definitions

A local definition of a community focuses solely on the community itself and possibly on its immediate neighbourhood. The rest of the graph is not considered. The following disposition sticks closely to that of Wasserman and Faust [37], whereas the presentation of the definitions also depends on [11]. Wasserman and Faust categorize what they call subgroups, and that we may call communities, into four categories: 1) communities based on complete mutuality, 2) communities based on reachability and diameter, 3) communities based on vertex degrees, and 4) communities based on comparison between inside and outside community ties.

Definitions based on complete mutuality

A community may be defined as a subgroup whose members all know each other. This is what is called a *clique* in graph theory, i.e. a maximal complete subgraph of at least three vertices. A triangle is the simplest example of a clique. This

definition is clearly too strict when we are dealing with subgraphs of, say, ten or more vertices. A subgraph consisting of ten vertices where there is an edge between every pair of vertices except one should intuitively be considered as a good community. Still it would not be considered as a community according to this definition. Moreover, the completely symmetric clique does not correlate with real world social networks where communities displays a hierarchy, with some people having a dominant position and others a more peripheral position.

Definitions based on reachability and diameter

It is useful to relax the notion of a clique and instead define communities that are almost cliques. A key idea is to include the property of paths between vertices in the definition, giving rise to definitions such as k-clique, k-clan, and k-club.

Luce [19] introduced the concept of an k-clique . A *k-clique* is a maximal subgraph in which no two vertices are at a distance more than k apart from each other, i.e. the number of edges in a shortest path between two arbitrary vertices is $\leq k$. This definition is, however, problematic because a shortest path may go through vertices that are not part of the subgraph under consideration. Thus, the diameter of the subgraph may exceed k, and the subgraph may even be disconnected, i.e. there exists two vertices in the subgraph such that there is no path between them only running through vertices inside the subgraph.

Mokken [22] resolved these problems with the introduction of the concepts k-clan and k-club. A *k-clan* is a k-clique with diameter at most k, i.e. a subgraph where the distance between any two vertices in the subgraph does not exceed k. A *k-club* is a maximal subgraph of diameter k.

Definitions based on vertex degrees

Another approach is based on the idea that in order for a vertex to be part of a community it must be adjacent to at least a certain number of other vertices in the community.

Seidman and Foster [33] introduced the concept of a *k-plex*, which is a maximal subgraph in which any vertex is adjacent to all other vertices of the subgraph except at most k of them. Seidman [32] also introduced the concept of a *k-core*, a maximal subgraph in which any vertex is adjacent to at least k other vertices. Fortunato [11] considers the k-core "essentially the same" as the p-quasi complete subgraph defined by Matsuda et al. [21]. This is a subgraph $QC = (V, E)$ such that $k_v \geq \lceil p(k-1) \rceil \forall v \in V$, where p and k denote the connectivity ratio of QC ($0 \leq p \leq 1$) and the number of vertices in the subgraph.

Definitions based on comparison between inside and outside community ties

It is only meaningful to speak of a subgraph as a community if the number of edges inside the subgraph is considerably greater than the number of edges

between the subgraph and the rest of the graph. This simple consideration has given rise to the fourth group of local definitions.

Luccio and Sami [18] introduced the concept of an *LS-set*, which is a subgraph $\mathcal{C} = (V, E)$ such that $k_v^{int(\mathcal{C})} > k_v^{ext(\mathcal{C})} \; \forall v \in V$. Radicchi et al. [28] calls this a *strong community* but they also define a *weak community* C such that $k_{int}^{\mathcal{C}} > k_{ext}^{\mathcal{C}}$. These two definitions will be central in chapter 8 and chapter 9, in the main part.

Borgatti et al. [2] defines the *edge connectivity* of two vertices to be the minimum number of edges one has to remove in order to disconnect them. A *lambda set* is then a subgraph such that any two vertices in the subgraph have greater edge connectivity than any pair of vertices where one belongs to the subgraph and the other not. The weakness of this definition is that two vertices in a lambda-set are not necessarily adjacent to each other.

3.3 Global definitions

There is a group of definitions that states that a graph displays community structure if it is different from a random graph in some way. A random graph is not expected to have community structure because there is an equal probability that any two vertices are adjacent. A *null model* of some kind is generated in order to compare it with the graph under consideration. The most famous null model is probably that of Girvan and Newman, which is a randomized version of the original graph such that the expected degree of every vertex is equal to the degree of the vertex in the original graph. This null model is the foundation of the important *modularity* function which will be discussed further in chapter 4.

3.4 Definitions based on vertex similarity

This section is based solely on the review article by Santo Fortunato called *Community detection in graphs* [11].

Definitions based on vertex similarity uses either local or global criteria to compute the similarity between every pair of vertices. This group of definitions is not important for the rest of this thesis and we will therefore not delve further into this subject, with the exception of one definition, given to wet the tongue of the reader. The *neighbourhood overlap* ω_{uv} between the neighbourhoods $N(u)$ and $N(v)$ of vertices u and v is defined as

$$\omega_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} \tag{3.1}$$

Given this definition we can compute the similarity of every pair of vertices in the graph. The closer the number is to 1, the more similar the vertices are.

Chapter 4

Modularity

In community detection we are given a graph and want to find a partition of its vertex set so that each class of the partition can be seen as a separate community.

This chapter introduces a function called modularity that is used to measure the quality of such a vertex set partition, when viewed as a set of communities. Unless otherwise stated, the material is based on the review article *Community detection in graphs* by Santo Fortunato [11], one of the major authorities in the field of community detection.

4.1 Partitions and quality functions

A partition of a graph is a division into disjoint communities such that every vertex is assigned one community. All partitions are not equally good and we need some way of ranking them. A *quality function* is a function that maps each partition of a graph to a number representing the quality of the partition. Higher numbers generally mean better partitions.

Most quality functions are additive. A quality function $\mathcal{Q}$ is *additive* if there exists a function f that can be applied to each community $\mathcal{C}$ of a partition $\mathcal{P}$ such that the quality of the partition is the sum of the qualities of the individual communities.

$$\mathcal{Q}(\mathcal{P}) = \sum_{\mathcal{C} \in \mathcal{P}} f(\mathcal{C}) \tag{4.1}$$

4.2 Introducing modularity

The most famous quality function is the *modularity* function of Newman and Girvan [24]. The basic idea behind the modularity function is to compare the edge density of a given subgraph (a community candidate) with the edge density of a randomized version of the same subgraph. The randomized version is not expected to have community structure. The randomized version of the subgraph
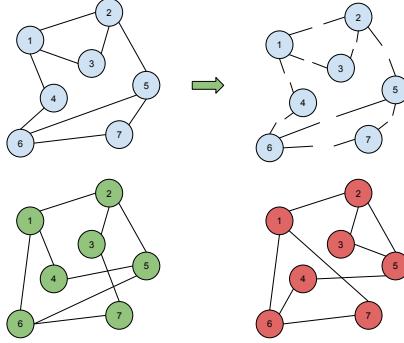
Figure 4.1: Slice all m edges of the original blue graph to obtain $2m$ edge stubs. The green and red graphs on the second line are the product of two possible randomized rewirings of the edges stubs of the blue graph. Note that each vertex keeps the same degree.

is part of a randomized version of the whole input graph called a *null model*, that keeps some of the structures of the original graph but that does not display community structure. The modularity function of Newman and Girvan does not depend on a specific null model. The modularity function $\mathcal{Q}$ is a sum that runs over all possible pairs of vertices.

$$\mathcal{Q}(\mathcal{C}) = \frac{1}{2m} \sum_{i \in V, j \in V} (A_{ij} - P_{ij}) \delta(\mathcal{C}_i, \mathcal{C}_j) \tag{4.2}$$

A is the adjacency matrix, with $A_{ij} = 1$ if vertices i and j are adjacent, m is the number of edges in the whole graph, P_{ij} is the expected number of edges between vertices i and j in the null model, $\mathcal{C}$ is a partition of the graph into communities, $\mathcal{C}_i$ and $\mathcal{C}_j$ are the communities of vertices i and j respectively, and $\delta(i, j) = 1$ if vertices i and j are in the same community ($\mathcal{C}_i = \mathcal{C}_j$), and 0 otherwise.

So what does this function say, intuitively? Given a partition into disjoint groups, if there are significantly more edges inside the groups than there would be between the same vertices in a randomized version of the graph, then we have a partition into real communities. The higher the value of $\mathcal{Q}$, the better the partition is.

There exists several null models. Fortunato claims that it is preferable to choose a null model in which the degree distribution is the same as in the original graph [11]. The standard null model of modularity is designed such that the expected degree sequence matches the actual degree sequence of the graph. In other words, the expected degree of a vertex v must match the actual degree of v. The expected degree sequence is computed as the average of all possible configurations of the model. This model is essentially equivalent to the *configuration model*, which works as follows. Conceptually slice each of the m edges of the graph into two stubs to obtain $2m$ stubs, see Figure 4.1. The

probability p_i of choosing one of the stubs of vertex i is $\frac{k_i}{2m}$ and the probability of making a connection between vertices i and j is $p_i p_j = \frac{k_i \cdot k_j}{4m^2}$. The expected number of edges between vertices i and j is then $P_{ij} = 2m p_i p_j = \frac{k_i \cdot k_j}{2m}$. The modularity function $\mathcal{Q}$ can then be written as:

$$\mathcal{Q}(\mathcal{C}) = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(\mathcal{C}_i, \mathcal{C}_j) \tag{4.3}$$

Only pairs of vertices that belong to the same community contribute to the sum of equation (4.3) and therefore the equation can be rewritten as follows.

$$\mathcal{Q} = \sum_{c=1}^{n_c} \left(\frac{l_c}{m} - \left(\frac{k^c}{2m} \right)^2 \right) \tag{4.4}$$

where n_c is the number of communities, l_c is the total number of edges joining vertices in community c, and k^c is the sum of the degrees of the vertices of c. $\frac{l_c}{m}$ is the fraction of edges of the graph that are inside community c. $(\frac{k^c}{2m})^2$ is the expected fraction of edges if the graph was a random graph such that the expected degrees of two arbitrary vertices were the same.

Equation (4.4) states that the quality of a community increases when the positive difference between the actual and the expected number of internal edges of a community increases. The modularity of the whole graph is zero, it is always smaller than one, and it can also take negative values. The modularity of a partition where each vertex constitutes its own community is negative because the first term of equation (4.4) then is zero. This implies that a graph without any partition of positive modularity does not display community structure.

The maximum modularity is, however, dependent both on the size of the graph and on the number of well-defined communities.

Modularity has been used as a quality function in many algorithms, including some of the algorithms that we will discuss in section 5.2.2 on divisive algorithms. Modularity optimization is itself a community detection method, which will be discussed in section 5.3 on greedy modularity-based community detection methods.

The modularity function can easily be extended to graphs with weighted edges. The degrees of vertices i and j, k_i and k_j, must be replaced by their strengths, s_i and s_j, where the strength of a vertex is the sum of the weights of the edges adjacent to the vertex. The number of edges in the graph, m, must be replaced by the total edge weight, W, of the graph in order to provide a proper normalization. The product $s_i s_j / 2W$ is the expected weight of edge $\{i, j\}$ in the null model and this quantity has to be compared to its actual weight, $w(\{i, j\})$. Thus the modularity function for weighted graphs is as follows.

$$\mathcal{Q}_W(\mathcal{C}) = \frac{1}{2W} \sum_{ij} \left(w(\{i, j\}) - \frac{s_i s_j}{2W} \right) \delta(\mathcal{C}_i, \mathcal{C}_j) \tag{4.5}$$

4.3   The limitations of modularity

In this section we will explore the limitations of modularity. The first topic to be discussed is the maximum obtainable value of modularity. The second topic to be dealt with is the quality of the partitions found when using modularity as a quality function. When dealing with this topic we will explain a problematic concept called the resolution limit of modularity.

The maximum modularity value

The first issue to be discussed is the value of the maximum modularity $\mathcal{Q}_{max}$ for a graph. Since the partition with all vertices in one community has zero modularity, we know that $\mathcal{Q}_{max}$ is non-negative. However, a large value for $\mathcal{Q}_{max}$ does not necessarily imply that the graph has community structure. Guimerà et al. [15] have demonstrated that random graphs may have partitions with large modularity values due to variations in the edge distribution. The variations may produce clustering of edges in subsets of the graph resulting in structures with the appearance of communities.

The quality of the partitions found by modularity

Fortunato and Barthélemy [12] have drawn attention to the fundamental issue concerning the ability of modularity to find good partitions. Given a graph with an obvious community structure, one should expect that the value of $\mathcal{Q}_{max}$ should reveal this. Recall that the expected number of edges between vertices i and j is $P_{ij} = k_i k_j / 2m$. Likewise, the expected number of edges between two communities $\mathcal{A}$ and $\mathcal{B}$ with total degrees $k^{\mathcal{A}}$ and $k^{\mathcal{B}}$, respectively, is $P_{\mathcal{A}\mathcal{B}} = k^{\mathcal{A}} k^{\mathcal{B}} / 2m$, and the change in modularity when $\mathcal{A}$ and $\mathcal{B}$ are merged is $\Delta\mathcal{Q}_{\mathcal{A}\mathcal{B}} = \ell_{\mathcal{A}\mathcal{B}}/m - k^{\mathcal{A}} k^{\mathcal{B}} / 2m^2$, where $\ell_{\mathcal{A}\mathcal{B}}$ is the number of edges between the two communities. If there is only one edge between $\mathcal{A}$ and $\mathcal{B}$ ($\ell_{\mathcal{A}\mathcal{B}} = 1$), then the two communities are expected to be kept as separate communities. Fortunato and Barthélemy show that if the two communities have approximately the same number m' of edges, $m' < \sqrt{2m}$, and they are connected, then modularity is greater if they are merged, which is an unfortunate result, see [12] and [11]. To see how this is possible, consider two subgraphs both of small total degree. In this case it is possible that the expected number of edges in the null model may be smaller than one and so it suffices to have only one edge between them in order for them to be merged. In fact, it turns out that the subgraphs will be merged (since this increases modularity) independently of their structure, such that even complete subgraphs will be merged.

In figure 4.2 eight communities, each a complete graph on ℓ vertices, are connected in a circle such that every community is connected to exactly two other communities and every pair of connected communities is connected by a single edge. Intuitively a good community detection algorithm should find exactly eight communities in this graph but in fact it turns out that when the number of communities n_c is larger than about ℓ^2, where ℓ is the number
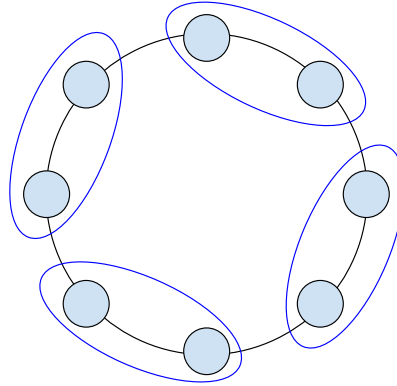
Figure 4.2: A circle of eight communities (the blue circles), each a complete graph on ℓ vertices, such that if two communities are connected, then they are connected by a single edge.

of vertices in each community, modularity will be higher for partitions where communities are merged.

This implies that modularity optimization has what is known as a *resolution limit* that may prevent community detection algorithms based on modularity optimization from detecting communities that are small compared to the whole graph, resulting in large communities swallowing up small communities or several small communities being lumped together. If the partition corresponding to $\mathcal{Q}_{max}$ has communities with total degree $\leq \sqrt{m}$, one cannot know beforehand whether the resulting communities are single communities or amalgamations of smaller weakly interconnected communities. In chapter 7 in the main part we will see this problem when we implement and test a greedy modularity optimization algorithm.

The resolution limit of modularity is due to the very definition of the null model of modularity, in particular the underlying assumption that every vertex can see every other vertex of the graph, in other words, that a vertex may be connected to any other vertex of the graph with equal probability. This is a wrong assumption especially for large social networks such as the Facebook graph, since it is more likely, say, for a Norwegian to get in touch with other Norwegians than with Chinese people.

One could imagine that a solution to the problem is to run a modularity optimization algorithm to obtain a partition of the input graph and then to run the same algorithm on each community that was found in the first round in order to find artificial amalgamations of communities. This is however not a fruitful approach since 1) the local modularities used to find partitions within the communities have different null models and are thus inconsistent, and 2) one has to design a stopping criterion for when to stop such a recursive approach, but no such criterion is obvious.

Good et al. [14] have discovered that the modularity landscape is character-
ized by an exponential number of distinct partitions whose modularity values all
are very close to the global maximum. This explains why many heuristics are
able to obtain a partition whose modularity is close to $\mathcal{Q}_{max}$ and it also implies
that the global maximum is practically impossible to find with anything other
than a brute-force search. However, even though two arbitrary partitions both
may have modularity values close to $\mathcal{Q}_{max}$, they are not necessarily structurally
similar to each other. Furthermore, the best partition from a structural view-
point is usually not the same as the partition that corresponds to $\mathcal{Q}_{max}$, but
it may have a high modularity value. So to summarize, the structurally best
partition is almost impossible to find in the midst of a plethora of partitions of
high modularity that are structurally different from the best one.

4.4   Motif modularity

In this section we will present one alternative to the standard modularity func-
tion. The alternative is called motif modularity and it has inspired a divisive
algorithm that we will implement and test in chapter 8 in the main part.

Arenas et al. [1] have introduced the concept of *motif modularity*. A motif
$\mathcal{M} = (V_{\mathcal{M}}, E_{\mathcal{M}})$ is an induced subgraph of a certain kind, e.g. a triangle or
a C_k ($k \geq 0$). See Figure 4.3(a) for some examples. Given an unweighted
undirected graph and a partition $\mathcal{C}$ of it into communities, the number of motifs
fully included inside communities is given by

$$\psi_{\mathcal{M}}(\mathcal{C}) = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \cdots \sum_{i_M=1}^{n} \prod_{(a,b) \in E_{\mathcal{M}}} w_{i_a} w_{i_b} \delta(\mathcal{C}_{i_a}, \mathcal{C}_{i_b}) \qquad (4.6)$$

(a) Some motifs: C_3, C_4, C_8, and P_8 (b) Two degenerated C_8 motifs.

Figure 4.3

The vertices of the motifs are labeled by the indices $i_1, i_2, ..., i_M$. $w_{i_a} w_{i_b}$
is equal to 1 if there is an edge between vertices i_a and i_b and 0 otherwise.
$\delta(\mathcal{C}_{i_a}, \mathcal{C}_{i_b})$ is equal to 1 if vertices i_a and i_b belong to the same community and
0 otherwise. It should be noted that this sum includes degenerated motifs, that
is, motifs where a single vertex is counted more than once. See Figure 4.3(b).

The maximum value of $\psi_{\mathcal{M}}$ corresponds to the partition consisting of a single community containing all the vertices:

$$\psi_{\mathcal{M}} = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \cdots \sum_{i_M=1}^{n} \prod_{(a,b) \in E_{\mathcal{M}}} w_{i_a} w_{i_b} \tag{4.7}$$

In a random undirected graph where the degrees of the vertices are preserved, the number of motifs fully included inside communities is

$$\Omega_{\mathcal{M}}(\mathcal{C}) = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \cdots \sum_{i_M=1}^{n} \prod_{(a,b) \in E_{\mathcal{M}}} k_{i_a} k_{i_b} \delta(\mathcal{C}_{i_a}, \mathcal{C}_{i_b}) \tag{4.8}$$
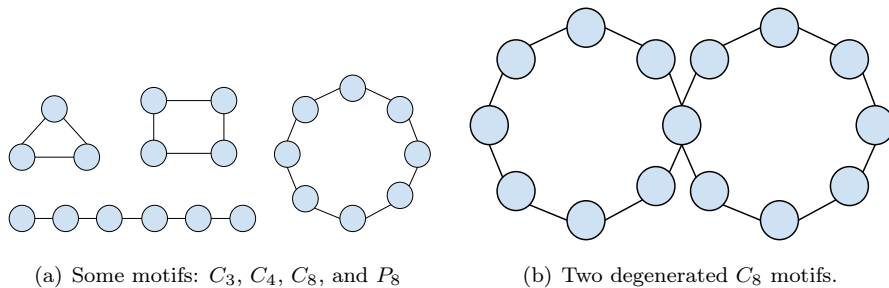
The maximum value of $\Omega_{\mathcal{M}}$ is given by

$$\Omega_{\mathcal{M}} = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \cdots \sum_{i_M=1}^{n} \prod_{(a,b) \in E_{\mathcal{M}}} k_{i_a} k_{i_b} \tag{4.9}$$

Given these quantities motif modularity is defined as the fraction of motifs inside the communities minus the fraction of motifs inside the communities of a random graph where the degrees of the vertices are preserved:

$$Q_{\mathcal{M}}(\mathcal{C}) = \frac{\psi_{\mathcal{M}}(\mathcal{C})}{\psi_{\mathcal{M}}} - \frac{\Omega_{\mathcal{M}}(\mathcal{C})}{\Omega_{\mathcal{M}}} \tag{4.10}$$

To simplify the expression above, Arenas et al. introduces the following three quantities.

- $n_{ij} = k_i k_j$ is the nullcase weight

- $w_{ij}(\mathcal{C}) = w_{ij} \delta(\mathcal{C}_i, \mathcal{C}_j)$ is the masked weight

- $n_{ij}(\mathcal{C}) = n_{ij} \delta(\mathcal{C}_i, \mathcal{C}_j)$ is the masked nullcase weight

Using these quantities $\mathcal{Q}_{\mathcal{M}}(\mathcal{C})$ can be rewritten as

$$\mathcal{Q}_{\mathcal{M}}(\mathcal{C}) = \frac{\sum_{i_1 i_2 \cdots i_M} \prod_{(a,b) \in E_{\mathcal{M}}} w_{i_a i_b}(\mathcal{C})}{\sum_{i_1 i_2 \cdots i_M} \prod_{(a,b) \in E_{\mathcal{M}}} w_{i_a i_b}} - \frac{\sum_{i_1 i_2 \cdots i_M} \prod_{(a,b) \in E_{\mathcal{M}}} n_{i_a i_b}(\mathcal{C})}{\sum_{i_1 i_2 \cdots i_M} \prod_{(a,b) \in E_{\mathcal{M}}} n_{i_a i_b}} \tag{4.11}$$

Using $\mathcal{Q}_{\mathcal{M}}(\mathcal{C})$ as a starting point, several subclasses of motif modularity can be defined. *Triangle modularity* $\mathcal{Q}_{\triangle}(\mathcal{C})$ is based on the triangle motif $E_{\triangle} = \{(1,2),(2,3),(3,1)\}$:

$$\mathcal{Q}_{\triangle}(\mathcal{C}) = \frac{\sum_{ijk} w_{ij}(\mathcal{C}) w_{jk}(\mathcal{C}) w_{ki}(\mathcal{C})}{\sum_{ijk} w_{ij} w_{jk} w_{ki}} - \frac{\sum_{ijk} n_{ij}(\mathcal{C}) n_{jk}(\mathcal{C}) n_{ki}(\mathcal{C})}{\sum_{ijk} n_{ij} n_{jk} n_{ki}} \tag{4.12}$$

Arenas et al. have also defined a more general cycle modularity and path modularity, and they have also devised definitions that apply to both directed and weighted graphs, but we will not look any further at these definitions as they are not important for the rest of this thesis.

Chapter 5

Community detection algorithms

In this section we will present three different approaches or methods that can be used to find clusters in complex networks. The first one, graph partitioning, is the problem of partitioning a graph into a predefined number of smaller components with specific properties. The second one, hierarchical clustering, is a set of algorithms that can reveal the hierarchy of communities in complex networks. Finally, modularity-based algorithms is a set of algorithms that seek to maximize the modularity function. We will only investigate greedy algorithms that maximize modularity.

The algorithms that will be presented in this chapter are given in Table 5.1. Algorithms that build upon each other are grouped together.

This chapter is based on the review article *Community detection in graphs* by Santo Fortunato [11], one the major authorities in the field of community detection.

Section	Year	Short name	Authors	Type	Parent	Is implemented?
section 5.1, section 5.3.3	1970	KL	Kernighan-Lin [16]	GP		No
section 5.2.2, section 8.1.1, section 8.1.2	2002	GN	Michelle Girvan and Mark Newman [13] [24]	DH		No
section 5.2.2, chapter 8	2004	REB	Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, Domenico Parisi [28]	DH	GN	No
section 5.2.2, chapter 8, chapter 9	2004	RECC	Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, Domenico Parisi [28]	DH	GN	Yes
section 5.3.1, section 5.3.2, section 5.3.3	2004	N	Mark E. J. Newman [23]	GAH		No
section 5.3.2, section 5.3.3, chapter 7, chapter 9	2004	CNM	Aaron Clauset, Mark E. J. Newman, and Cristopher Moore [5]	GAH	N	Yes
section 5.3.3	2007	WT	Ken Wakita and Toshiyuki Tsurumi [35]	GAH	CNM	No
section 5.3.3	2008	SC	Philipp Schuetz and Amedeo Caflisc [30]	GAH	CNM, KL	No

Table 5.1: The algorithms studied in this chapter. The column "Section" gives an overview of the sections of this thesis where the given algorithms are mentioned. The following abbreviations are used in the column "Type", which explains what type of algorithm the given algorithm is. GP = graph partitioning, DH = divisive hierarchical, and GAH = greedy agglomerative hierarchical. The "Parent" column contains the algorithm(s) that the given algorithm builds upon. The column "Is implemented?" contains "yes" if the algorithm is implemented by the author of this thesis, and "no" otherwise.

5.1 Graph partitioning

Graph partitioning is the problem of partitioning a graph into a predefined number of smaller components with specific properties. A common property to be minimized is called *cut size*. A cut is a partition of the vertex set of a graph into two disjoint subsets and the size of the cut is the number of edges between the components. A multicut is a set of edges whose removal leaves a graph of two or more components. When one is doing graph partitioning it is necessary to specify the number of components one wishes to get and the size of the components. Without a requirement on the number of components, the solution would be trivial, just put all the vertices into one component to obtain a cut size of zero. The size of the components must also be specified, as otherwise a likely but not meaningful solution would be to put the minimum degree vertex into one component and the rest of the vertices into another component.

The *Kernighan-Lin algorithm* is a $\mathcal{O}(n^3)$ heuristic algorithm and one of the earliest algorithms proposed to solve the graph partitioning problem. The original motivation was to assign the "components of electronic circuits to circuit boards to minimize the number of connections between boards" [16]. The algorithm maximizes the difference $\mathcal{Q}$ between the number of edges inside the components and the number of edges between the components. Initially the algorithm partitions the graph into two components of a predefined size. This may be done at random or by using some properties of the graph. Then a number of swaps of pairs of vertices are made until some maximum of $\mathcal{Q}$ is reached. The algorithm also makes use of some swaps that decrease $\mathcal{Q}$ in order to escape local maxima.

We will not look at other graph partitioning algorithms because they are usually not suitable to do community detection with. Since the number of communities in a social network usually is not known in advance, graph partitioning methods are not fit to detect communities in such networks because they require as input a number specifying the number of partitions to be output.

5.2 Hierarchical clustering

An interesting property of social networks is that they often exhibit a hierarchical structure with smaller communities dwelling inside larger communities. This fact justifies a special group of algorithms called *hierarchical community detection algorithms* that can reveal the hierarchy of communities in social networks. The hierarchical community detection algorithms are usually divided into two groups:

1 *Agglomerative hierarchical algorithms*

2 *Divisive hierarchical algorithms*

Agglomerative algorithms are bottom-up algorithms in which two communities at a time are merged if they are sufficiently similar to each other. These

algorithms starts with each vertex as a community on its own and ends up with the whole graph as one community if they are not stopped at some earlier stage. Divisive algorithms, on the other hand, are top-down algorithms in which communities are iteratively split into two parts by removing edges between the least similar vertices. The skeleton structures of the two algorithmic approaches are given in Algorithm 1 and Algorithm 2, and schematic figures of the two approaches are given in Figure 5.1(a) and Figure 5.1(b).
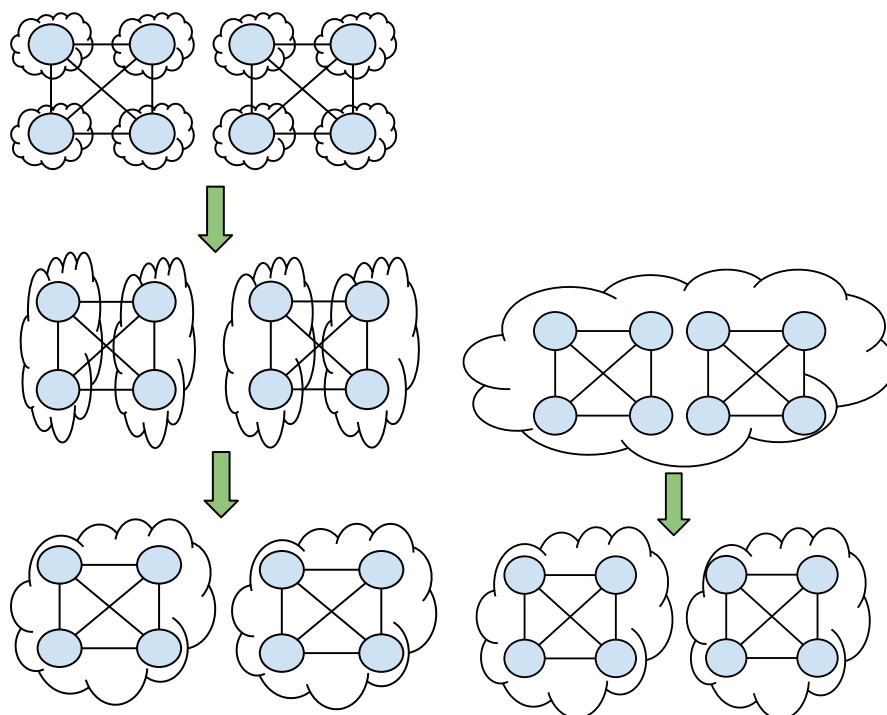
(a) A schematic figure of an agglomerative algorithm.   (b) A schematic figure of a divisive algorithm.

Figure 5.1: Schematic figures of agglomerative and divisive algorithms.

Both categories of algorithms have in common that they initially compute a similarity matrix X based on some similarity measure, e.g. one of those mentioned in section 3.4. The similarity of every pair of vertices is computed, also pairs of vertices not connected by an edge.

Although hierarchical community detection have some advantages over graph partitioning it also has several disadvantages. The method does not distinguish between the many communities that it finds, some of which are artificial and some of which are in fact meaningful. The method tends to place some vertices outside their communities, even vertices with an obvious central role in their respective communities [25]. They also tend to make a one man community out
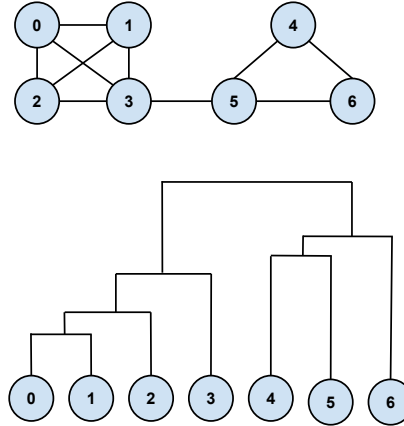
Figure 5.2: The figure shows a graph at the top and at the bottom a dendrogram showing how vertices 0 and 1 first are merged into a community, which then absorbs vertices 2 and 3. Next vertices 4 and 5 are joined into another community, which then absorbs vertex 6. Finally communities $\{0, 1, 2, 3\}$ and $\{4, 5, 6\}$ are joined into a single community $\{0, 1, 2, 3, 4, 5, 6\}$.

of vertices with a single neighbour, which usually doesn't make sense in social networks.

The complete hierarchy of partitions found by a hierarchical algorithm can be neatly illustrated by a *dendrogram*, as illustrated in Figure 5.2. The dendrogram shows successive joins done by the algorithm from the bottom and upwards. Slicing the dendrogram horizontally at different levels yields the different partitions of the graph. The closer to the bottom of the dendrogram the horizontal slicing is done, the higher the number of communities of the corresponding partition is. The first join made in the algorithm is shown as a line at the bottom of the dendrogram connecting two vertices and the last join made by the algorithm is shown as the uppermost line connecting either two vertices, two other lines, or one vertex and one line. Thus a dendrogram yields more information than a simple rooted tree, as found in graph theory.

5.2.1 Agglomerative hierarchical algorithms

Several agglomerative algorithms run in $\mathcal{O}(n^3)$ time. As mentioned a similarity matrix X is computed, containing the similarity values for every pair of vertices, either they are connected by an edge or not. At each iteration of the algorithm it merges the two most similar communities. Three different approaches in how to find the two most similar communities exist.

The *single linkage clustering* method [39] picks at each iteration the two communities with the smallest element x_{ij} of the similarity matrix X, such that vertex i is in one community and vertex j in the other community.

Agglomerative algorithm 1: A generic agglomerative algorithm
---
    input : An input graph G
    output: A partition into communities

    Compute similarity of every vertex pair;
    Put every pair of vertices with corresponding similarity value into a
    priority queue Q;
    Start with every vertex as a community on its own;
    while *stopping condition not met* **do**
        if *Q is empty* **then**
            break;
        if *current partition satisfies the specified requirements* **then**
            break;
        Remove the most similar pair of vertices from Q and merge their
        corresponding communities into a single community;
    Return the computed partition into communities;

The *complete linkage clustering* method does the quite opposite. At each iteration it picks the maximum element x_{ij} of X and merges the communities of vertices i and j, see [38] and [34].

The *average linkage clustering* method computes the average of all similarity values.

5.2.2 Divisive hierarchical algorithms

The basic idea behind divisive hierarchical algorithms is to detect and then remove edges between vertices of different communities in order to separate the communities from each other. Initially there is one community containing all the vertices of the graph.

The Girvan-Newman algorithm

The most well-known divisive algorithm is the algorithm of Girvan and Newman [13] [24], presented in Algorithm 3. The key idea of the algorithm is to find the edges that are most clearly candidates for being "between" communities, that is, to find good inter-community edges. The underlying concept is called *edge betweenness* and is defined to be the number of shortest paths between pairs of vertices that go along the edge. If there are several shortest paths between a pair of vertices, each path is given an equal weight. Edges between communities will therefore have the highest edge betweenness values.

The algorithm keeps removing edges until none are left. At each iteration the edge with the highest edge betweenness value is removed. As with all hierarchical algorithms, the sequence of edge removals defines a dendrogram describing

---

Algorithm 2: A generic divisive algorithm

input : An input graph G
output: A partition into communities

Compute similarity of every vertex pair
Put every pair of vertices with corresponding similarity value into a
priority queue Q
Start with all vertices as a single community
while *stopping condition not yet met* **do**

    if *Q is empty* **then**
        break

    if *current partition satisfies the specified requirements* **then**
        break

    Remove the least similar pair of vertices from Q and remove the edge
    between them

Return the computed partition into communities

---

the hierarchy of partitions into communities. The algorithm returns the partition with the highest modularity value.

Algorithm 3: The divisive algorithm of Girvan and Newman

input : A graph $G = (V, E)$, where $E = \{e_1, ..., e_m\}$
output: A partition of G into communities
Calculate the betweenness values of all edges;
while *more edges left* **do**

    Remove the edge with the greatest betweenness value;
    Recalculate betweenness values for all edges affected by the removal;
    Compute the modularity value of the current partition;

Return the partition with the highest modularity value;

The betweenness of all edges can be calculated in $\mathcal{O}(mn)$ time on unweighted graphs, or $\mathcal{O}(n^2)$ on unweighted sparse graphs, and $\mathcal{O}(nm+n^2 log\ n)$ on weighted graphs, with techniques based on breadth-first search [3].

A downside of the algorithm is that after calculating all the edge betweenness values, if there are more than one edge between two communities, then only one of them is guaranteed to have a high edge betweenness value. After removing an edge of highest betweenness value one therefore has to recalculate all the betweenness values in order for another inter-community edge between the same two communities to exhibit a high value.

Several improvements of the algorithm of Girvan and Newman have been proposed but we will not look any further at them.

Other divisive algorithms

Many other divisive algorithms have been proposed and they use different functions to detect the edges to be removed. In chapter 8 in the main part we will implement and test a divisve algorithm that uses the concept of an *edge-clustering coefficient* to find and remove inter-community edges [28].

5.3 Greedy modularity based community detection

Modularity is the most used quality function, see chapter 4. There are several techniques based on modularity, including greedy techniques, which we will explore in this section, simulated annealing, extremal optimization, and spectral optimization. The last three techniques are vast fields and quite different from the focus area of this thesis and will therefore not be explored any further.

5.3.1 Newman's algorithm

In 2004 Newman published an algorithm [23] that greedily maximizes the modularity function $\mathcal{Q}$, which is the fraction of edges that exist inside communities minus the expected value of the same quantity if the edges are spread randomly across the graph. This is meaningful since the greater the value of $\mathcal{Q}$ is, the better community structure the graph has. The algorithm falls within the general category of agglomerative hierarchical algorithms, that we discussed in section 5.2.1.

The algorithm starts with every vertex being a community of its own. Thus, given an input graph on n vertices, we initially have n communities. In $n - 1$ iterations the algorithm joins pairs of communities, at each iteration the pair that gives the highest increase or lowest decrease of $\mathcal{Q}$ is joined. The process can be represented by a dendrogram that shows the order of the joins. Cuts through the dendrogram at different levels give different partitions into communities. The best cut can be found by taking the partition that gives the highest value of $\mathcal{Q}$. See Figure 5.2 for an example of a dendrogram.

Newman maintains a matrix e that for element e_{ij} stores the fraction of edges in the network that connect vertices in community i to those in community j, and an array a such that $a_i = \sum_i (e_{ii} - a_i^2)$. The change in $\mathcal{Q}$ when joining communities i and j is given by $\Delta \mathcal{Q} = e_{ij} + e_{ji} - 2a_i a_j = 2(e_{ij} - a_i a_j)$, which can be computed in constant time.

At a given iteration there are at most m possible community joins, m being the number of edges in the graph. For each possible join $\Delta \mathcal{Q}$ is computed and the join that gives the highest increase or lowest decrease of $\mathcal{Q}$ is picked. When joining two communities i and j some elements of the matrix e must be updated by adding together the rows and columns corresponding to the two joined communities. For instance, if communities i and j are to be joined and there are 2 edges between communities i and k ($e_{ik} = 2/m$) and 3 edges between

communities j and k ($e_{jk} = 3/m$), and the new community is called i, then e_{ik} must be updated to the value $5/m$. Updating the matrix e in a single iteration takes $\mathcal{O}(n)$ time. Thus finding the best join and then updating the matrix e takes $\mathcal{O}(n + m)$ time. The algorithm performs $n - 1$ joins altogether and hence the algorithm runs in $\mathcal{O}(n(n + m))$ time or $\mathcal{O}(n^2)$ time on sparse graphs.

5.3.2 The algorithm of Clauset, Newman and Moore

The improvements

In 2004 Clauset, Newman and Moore introduced an improvement [5] of the greedy algorithm that Newman published earlier the same year [23]. Whereas the original algorithm runs in time $\mathcal{O}((n + m)m)$, the improvement runs in time $\mathcal{O}(m \cdot d \cdot \log \mathrm{n})$, where d is the depth of the dendrogram describing the hierarchy of partitions. (In the following the algorithm by Clauset, Newman and Moore will be referred to as the CNM algorithm.) The improvement was based on the observation that updating the adjacency matrix e in Newman's algorithm involved many unnecessary operations due to the sparseness of the matrix. Clauset et al. maintains a sparse matrix $\Delta \mathcal{Q}$ containing only values for connected communities. $\Delta \mathcal{Q}_{ij}$ contains the increase or decrease in modularity if communities i and j are merged. Each row of the matrix is stored both as a binary tree, in order to find and insert elements in $\mathcal{O}(\log \mathrm{n})$ time, and as a max heap, such that the maximum element of a row can be found in constant time, that is, such that given a community i that is to be merged with some other community, one can find the neighboring community of i whose amalgamation with i will result in the highest increase (or lowest decrease) in modularity. Henceforth the version of $\Delta \mathcal{Q}$ being an array of binary trees is referred to as $\Delta \mathcal{Q}^t$ and the version being an array of max heaps is referred to as $\Delta \mathcal{Q}^h$. When something applies to both structures we will simply say $\Delta \mathcal{Q}$. Note that this notational distinction is invented by the author of this thesis and is not found in the paper of Clauset et al. Furthermore, yet another max heap H is maintained that stores the maximum element from every row of $\Delta \mathcal{Q}$. See Figure 5.3 for an illustration of the data structures used by the algorithm.

Overview of the algorithm

The algorithm starts with every vertex of the input graph G being a community of its own. Thus there are initially n communities. At every iteration, the maximum element from the max heap H is chosen if there still is more than one community. Thus a total of $n-1$ join operations will be performed. The element of H is a triple $(i, j, \Delta \mathcal{Q}_{ij})$ consisting of the labels of the two communities (i and j) whose amalgamation will result in the highest increase, $\Delta \mathcal{Q}_{ij}$, of modularity $\mathcal{Q}$ (or lowest decrease if no increase is possible). Merging i and j includes updating $\Delta \mathcal{Q}$, both the binary trees and the max heaps of rows i, j, and every neighbor k of either i or j or both, the overall max heap H with respect to i, j and every neighbor k, and the array a.
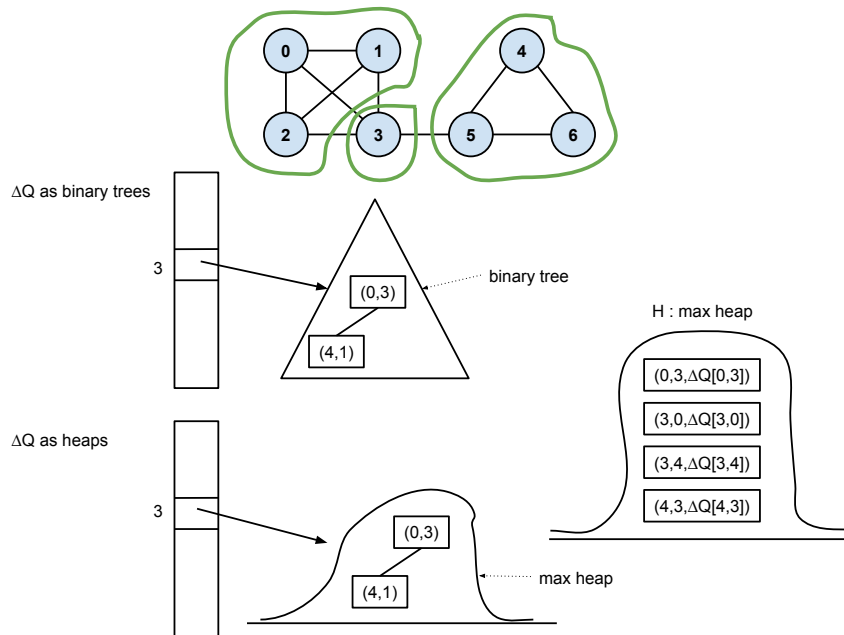
Figure 5.3: The CNM algorithm uses three main data structures. The $\Delta\mathcal{Q}$ values are stored both as an array of binary trees and as an array of max heaps. Furthermore, a max heap H stores the maximum element from every row of $\Delta\mathcal{Q}$. In this example we have a graph on seven vertices in the middle of the community detection process. The graph is so far partitioned into three communities $\{0, 1, 2\}$, $\{3\}$, and $\{4, 5, 6\}$. Not all details of the data structures are shown. We can see the row of community $\{3\}$ in both versions of $\Delta\mathcal{Q}$ pointing to a binary tree and a max heap, respectively. The element $(0, 3)$ in these data structures tells us that community $\{3\}$ is connected to community $\{0, 1, 2\}$ (indexed by the lowest numbered vertex in this community) by 3 edges. Since community $\{3\}$ is connected to community $\{4, 5, 6\}$ with only one edge, element $(0, 3)$ is the top element in the max heap and accidentally also in the binary tree. The max heap H stores the maximum element from every row of $\Delta\mathcal{Q}$.

Updating the data structures

At a given iteration communities i and j are merged and the new community is labeled as j. Thus row j must be updated in both versions of $\Delta\mathcal{Q}$ whereas row i can be removed. There are three update rules when updating row j.

1. If k is a common neighbor of i and j, then the element of k in row j is updated according to $\Delta\mathcal{Q}'_{jk} := \Delta\mathcal{Q}_{ik} + \Delta\mathcal{Q}_{jk}$.

2. If k is a neighbor only of i, then we must add an element of k in row j with the value $\Delta\mathcal{Q}'_{jk} := \Delta\mathcal{Q}_{ik} - 2a_j a_k$.

3. If k is a neighbor only of j, then the element of k in row j is updated according to $\Delta\mathcal{Q}'_{jk} := \Delta\mathcal{Q}_{jk} - 2a_i a_k$.

In the following we will let $|i|$ and $|j|$ denote the numbers of neighboring communities of i and j, respectively.

Implementing the three rules above requires scanning rows i and j in both $\Delta\mathcal{Q}^t$ and $\Delta\mathcal{Q}^h$. A single application of any of the three rules takes $\mathcal{O}(\log|j|) = \mathcal{O}(n)$ time both for $\Delta\mathcal{Q}^t$ and $\Delta\mathcal{Q}^h$. Thus it takes $\mathcal{O}(2(|i|+|j|)\log n) = \mathcal{O}((|i|+|j|)\log n)$ time altogether.

We change the maximum element of at most $|i|+|j|$ rows of $\Delta\mathcal{Q}$ and we therefore must do at most $|i|+|j|$ updates of the max heap H, each taking $\mathcal{O}(\log n)$ time.

$a'_j = a_j + a_i$ is updated in constant time.

Time complexity

As discussed in the previous subsection a single join operation takes $\mathcal{O}((|i|+|j|)\log n)$ time. Since the total degree of all the vertices in the graph is $2m$, $|i|+|j|$ is at most $2m$. If the depth of the dendrogram describing the hierarchy of communities is d, then then running time of the algorithm is $\mathcal{O}(md\log n)$.

A downside of the CNM algorithm

A downside of the CNM algorithm is that it tends to create large communities in general, even when the communities apparently should have smaller sizes, and hence gives poor values of modularity. The problem is due to the modularity function itself. See section 4.3 for a further discussion on the quality of the partitions found by the modularity function.

5.3.3 Improvements of the CNM algorithm

Several improvements of the CNM algorithm have been proposed, of whom we will mention only a few.

Danon et al. [7] have proposed to normalize the $\Delta\mathcal{Q}$ values in order to treat all communities as equal. Each $\Delta\mathcal{Q}_{ij}$ is normalized by dividing it on a_i, the fraction of edge stubs connected to vertices in community i. This implies that

$\Delta\mathcal{Q}_{ij} \neq \Delta\mathcal{Q}_{ji}$, but the modified algorithm takes both values into consideration when deciding which join operation to perform next. Another immediate consequence is that the communities with the smallest amount of edge stubs have the highest $\Delta\mathcal{Q}$ values and are thus joined first. The algorithm finds better communities than that of Newman, especially when the sizes of the communities vary a lot.
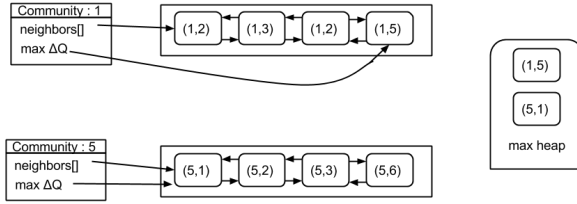

Figure 5.4: An example showing the data structures used in the algorithm of Wakita and Tsurumi.

Wakita and Tsurumi [35] have later showed that the CNM algorithm in practice is not as fast as Clauset et al. argue . Clauset et al. argue that d, the depth of the dendrogram, grows as log n for graphs that display a significant hierarchical structure, and that these graphs are sparse, such that the running time for such graphs is actually $\mathcal{O}(n\ log^2 n)$. However, Wakita and Tsurumi have showed that since the CNM algorithm favors large communities over small communities, the relation $d \sim log\ n$ does not hold and the running time tends toward the worst time complexity in practice, i.e. $\mathcal{O}(n^2\ log\ n)$. To remedy the unbalanced merge process they introduce the concept of *consolidation ratio* of a community amalgamation, defined as $ratio(c_i, c_j) = min\{|c_i|/|c_j|, |c_j|/|c_i|\}$. The size of community c_i, $|c_i|$, is defined as the number of edges going to other communities. Whereas the CNM algorithm considers all $\Delta\mathcal{Q}_{ij}$ values when it is to decide the next amalgamation, the algorithm of Wakita and Tsurumi considers $\Delta\mathcal{Q}_{ij} \cdot ratio(c_i, c_j)$. Furthermore, they make some changes to the data structures to further improve the running time. See Figure 5.4 for an example of the data structures used by Wakitra and Tsurumi.

- The balanced binary trees and max-heaps of the CNM algorithm are replaced by a doubly linked lists sorted by community ID.

- Each community c_i is represented by a data structure storing a list of all neighboring communities. The list actually consists of community pairs, i.e. pairs of numbers, such that if community 1 is linked to communities 3, 7 and 9, the list is $\{(1,3),(1,7),(1,9)\}$.

- The community data structure also holds a maximum pointer to the maximum community pair, i.e. a pair telling us which neighbor of community c_i will increase modularity the most if merged with c_i.

- A max-heap storing the maximum community pair of every community is maintained.

Maintaining pointers to the maximum community pair for all communities needs some care. Say two communities, c_i and c_j, are merged at a certain iteration of the algorithm, and among all the community pairs that have their $\Delta\mathcal{Q}$ values updated, community pair $p = (c_i, c_k)$ is one of them. If p is not the largest community pair in the list of c_k and the corresponding $\Delta\mathcal{Q}$ decreases, nothing has to be done. If on the other hand $\Delta\mathcal{Q}$ increases such that it becomes greater than the current maximum pair of c_k, then the maximum pointer must be updated to point to p. Furthermore, if p is the maximum community pair of c_k and its $\Delta\mathcal{Q}$ value increases, then no work is needed. If on the other hand $\Delta\mathcal{Q}$ decreases in this case, the whole list of c_k must be scanned in $\mathcal{O}(m)$ time in order to update the maximum pointer, which still may point at p. A worst case may happen if this last scenario occurs most of the time. However, Wakita and Tsurumi argue that if the given network adhere to the preferential attachment law, then there will probably be a heavily linked pair in most of the lists that the other pairs cannot compete with, and thus they will "hopefully" get fast updates of the $\Delta\mathcal{Q}$ values.

Another solution to avoid the formation of large communities absorbing smaller ones has been presented by Schuetz and Caflisch [30]. They have made a modified version of the CNM algorithm in which several community amalgamations may be done in a single iteration. The effect of this modification is several community centres emerging from the very beginning and growing simultaneously into larger communities. The algorithm is accompanied by a refinement procedure named "vertex mover" (VM) that is somewhat similar to the Kernighan-Lin algorithm described in section 5.1, the difference being that the vertex mover has a completely local focus. The VM is started when the algorithm converges toward some value. The procedure handles all vertices in increasing order of degree and reassigns every vertex to the neighboring community that gives the maximal modularity improvement. The procedure is repeated until no further improvements are possible.

Part II

Main part

Chapter 6

Introduction to the main part

In the main part we will describe implementations of two hierarchical clustering algorithms and present a community detection web service.

In chapter 7 we describe the implementation of the agglomerative hierarchical clustering algorithm by Clauset, Newman and Moore, that was presented in section 5.3.2. The algorithm will later on be referred to as the CNM algorithm. The algorithm greedily maximizes the modularity function described in chapter 4. The chapter is concluded with some tests on benchmark graphs. The results are not altogether good partly due to the simplicity of the modularity function, partly due to some inherent weaknesses of the modularity function, and partly due to the fact that there usually does not exist a partition of the vertex set of the input graph into communities that can be considered a final and definitive answer.

In chapter 8 we describe the implementation of a divisive hierarchical clustering algorithm by Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Paris [28]. This algorithm makes use of a clustering coefficient called the edge clustering coefficient (ECC) to detect and possibly remove edges that are good candidates to be inter-community edges. As increasingly more edges are removed from the graph, the graph is split into several connected components and when the algorithm finally ends, the vertices of each connected component is considered as a separate community. This algorithm is henceforth referred to as the RECC algorithm.

The RECC algorithm is not tested on benchmark graphs as was the CNM algorithm. Instead, and in response to a challenge posed by Radicchi et al. [28] that the results of a community detection algorithm on a given network needs to be interpreted in order to receive a final evaluation we describe the implementation of a community detection web service in chapter 9 that offers community detection on the collaboration networks in the DBLP database using the RECC algorithm.

Both the paper describing the CNM algorithm and the paper describing the RECC algorithm lack fine algorithmic details as they only give high level descriptions of the respective algorithms. This point has to be stressed when

the reader is considering the implementations described in this paper. Due to the high level descriptions of the algorithms there may be and most likely is differences between the implementations of this author and the implementations by the designers of the two algorithms. In fact, for each of the two algorithms, one can actually discern four different quantities to relate to.

- The mathematical description of the CNM and RECC algorithms in their original papers.

- The implementation of the CNM and RECC algorithms by their authors.

- The more fine-grained mathematical descriptions of the CNM and RECC algorithms offered by the author of this thesis.

- The implementations done by the author of this thesis based on his fine-grained description of the CNM and RECC algorithms.

This makes it quite likely that for instance the implementation of the CNM algorithm by Clauset et al. will not yield exactly the same results as this authors implementation on all instances. This fact can be observed when considering some iteration in the algorithm when there may be $k > 1$ community pairs on the max heap H (recall the details described in section 5.3.2) that all have equal modularity increase or decrease values if they are merged. Different heap implementations may pick different community pairs first and this can have an impact on the final community structure. Furthermore, it is not revealed in the CNM paper how rows i and j are searched when the corresponding communities are to be merged. This will also have an effect on the sequence of updates to both $\Delta \mathcal{Q}$ and H in a given iteration and thereby also to the configurations of these structures after the two communities are successfully merged.

The same argument can be given in case of the RECC algorithm. We can not rule out the possibility that different implementation details may have an impact on the final results.

6.1 Implementation environment

The CNM and RECC algorithms and the web service are implemented on an iMac with OX X Yosemite version 10.10.2, a 2.5 GHz Intel Core i5 processor, and with 4 GB 1333 MHz DDR3 as memory. All code is written in Java version 8 using the IDE Eclipse, Luna Service Release 2 (4.4.2). The web service is installed on a GlassFish Server Open Source Edition 4.1 application server and is located at the address www.dblpcommunities.com.

Chapter 7

The CNM algorithm

In the following chapter we will give a detailed description of a slightly modified version of the algorithm by Clauset, Newman and Moore, as described in section 5.3.2, including pseudo code, and descriptions both of the data structures used and of the time complexity. We will also show the results of running the algorithm on some benchmark graphs.

7.1 Implementing the algorithm

In this section we describe our own implementation of the CNM algorithm.

7.1.1 Differences between the implementations

There are some main differences between how the CNM algorithm is implemented and how the algorithm of this thesis is implemented.

In the following we refer the reader to the CNM algorithm description in section 5.3.2.

The CNM algorithm keeps choosing the best element from the heap H until H is empty and a single community remains. The author of this thesis has chosen to stop the algorithm when every possible remaining community amalgamation will result in a decrease in the modularity Q, reasoning that in many cases the a good enough result is obtained at this stage.

The second difference we will mention is that given two communities i and j that are merged, the new community will have as id $min\{i, j\}$. The paper of Clauset et al. does it the other way.

The third difference is a notational one. Both in the notation and in the implementation we make a clear distinction between the two versions of ΔQ, namely ΔQ^t and ΔQ^h, the first data structure being an array of binary trees and the second data structure being an array of indexed maximum heaps. These data structures are discussed in section 7.1.2.

The fourth difference is that our implementation has made an additional abstract layer consisting of a `Metagraph` class dealing with high level community information. No such thing is discussed in the CNM paper. We will discuss this data structure further in section 7.1.2.

There are probably several other differences between the two implementations but these are not immediate since the CNM paper is lacking of fine implementation details.

7.1.2 The Data Structures

In this subsection the main data structures used in the implementation of this author is explained in detail. First we will give a short overview of the data structures.

- $\mathbf{\Delta Q^t}$. An array of binary trees, more precisely an `ArrayList` of `TreeMap`s containing pairs of Integers and Doubles.

- $\mathbf{\Delta Q^h}$. An array of max heaps, more precisely an `ArrayList` of `IndexMaxPQ`s.

- **H**. A max heap, that is, an `IndexMaxPQ`.

- **G**. A `Graph`, storing the structural information of the input graph.

- **M**. A `Metagraph`, storing the high level community information, such as the number of edges internal to a community and the number of edges between two communities.

- **a**. An array holding the fraction of ends of edges that are attached to vertices in each community.

The sparse matrix $\mathbf{\Delta Q^t}$

$\mathbf{\Delta Q^t}$ is an `ArrayList` of `TreeMap`s. Both classes are part of the java.util package. The `TreeMap` is a Red-Black tree implementation based on algorithms in *Introduction to Algorithms* by Cormen, Leiserson, and Rivest. Table 7.1 shows the names and running times of some operations in the `TreeMap` class used by this algorithm. `navigableKeySet` runs in time $\mathcal{O}(deg(i)) = \mathcal{O}(n)$ for a community i and the running time is due to the fact that i may be a neighbor of $n-1$ other communities in the worst case. Operation `put` inserts a (key,value) pair to the tree. If the given key is already present in the tree, the previous value is replaced by the new one.

Table 7.1: **TreeMap<Key,Value>**

Method	Runtime
containsKey(Key)	$\mathcal{O}(1)$
Value **get**(Key)	$\mathcal{O}(\log n)$
NavigableSet<Key> **navigableKeySet**()	$\mathcal{O}(n)$
Value **put**(Key, Value)	$\mathcal{O}(\log n)$
Value **remove**(Key)	$\mathcal{O}(\log n)$

The sparse matrix ΔQ^h.

ΔQ^h is an `ArrayList` of `IndexMaxPQ`s. The latter class is an indexed maximum priority queue based on an implementation of Sedgewick and Wayne [31]. The queue is heap based. This data structure offers `contains`, `maxIndex`, and `maxKey` in constant time, whereas `deleteMax`, `changeKey`, `increaseKey`, `decreaseKey`, and `delete` are all performed in logarithmic time. Especially the operations offering some kind of change to the key associated with an index are useful since java.util.PriorityQueue does not offer such methods and the alternative would be to first delete a key and then to insert the increased or decreased key. The index feature is also helpful since this gives us the possibility of retrieving ΔQ_{ij}^h in constant time, although this is not strictly necessary since ΔQ_{ij}^t can be retrieved in constant time due to the underlying `TreeMap` and $\Delta Q_{ij}^t = \Delta Q_{ij}^h$. The methods and corresponding running times of `IndexMaxPQ` are shown in Table 7.2.

Table 7.2: **IndexMaxPQ<Key>**

Method	Runtime
changeKey(int, Key)	$\mathcal{O}(\log n)$
contains(int)	$\mathcal{O}(1)$
decreaseKey(int, Key)	$\mathcal{O}(\log n)$
delete(int)	$\mathcal{O}(\log n)$
deleteMax(): (int, Key)	$\mathcal{O}(\log n)$
increaseKey(int, Key)	$\mathcal{O}(\log n)$
insert(int, Key)	$\mathcal{O}(\log n)$
isEmpty(): boolean	$\mathcal{O}(1)$
keyOf(int): Key	$\mathcal{O}(1)$
maxIndex(): int	$\mathcal{O}(1)$
maxKey(): Key	$\mathcal{O}(1)$

The max heap H

H is an indexed maximum priority queue of type `IndexMaxPQ`. H stores the maximum element of every row of ΔQ. That is, for every community i the maximum element of ΔQ_i is stored in H and this value can be retrieved from ΔQ^h in constant time with operations `maxIndex` and `maxKey`. Since H is of

type `IndexMaxPQ`, it gives us the additional benefit of retrieving the maximum element of ΔQ_i in constant time by calling `keyOf`(i) on H, although this is superfluous since this value also can be retrieved in constant time by calling `maxKey` on ΔQ_i^t.

The graph G

G is a graph of type `Graph` and it stores structural information about the input graph and is never altered after it is constructed when reading the input. `Graph` is based on an implementation of Sedgewick and Wayne [31]. The methods and corresponding running times of *Graph* are shown in Table 7.3.

Table 7.3: **Graph**

Method	Runtime
empty(): boolean	$\mathcal{O}(1)$
numNodes(): int	$\mathcal{O}(1)$
numEdges(): int	$\mathcal{O}(1)$
addEdge(int,int): void	$\mathcal{O}(1)$
neighborhood(int): Iterable<Integer>	$\mathcal{O}(1)$
areNeighbors(int,int): boolean	$\mathcal{O}(\log n)$
numNeighbors(int): int	$\mathcal{O}(1)$

The metagraph M

M is implemented as the `Metagraph` class and the core of this class is a slightly modified union-find structure using path compression. This structure offers two methods, `find` and `union`, both running in $\Theta(1)$ amortized time. `Metagraph` maintains an `ArrayList` of `HashMap`s called `neighbors`, storing information about the neighboring communities of every community, an array `parent` holding the community ID of every vertex in the input graph G, and an array `degree`, storing the degree of every community. Here the degree of a community i is the number of ends of edges attached to vertices in i, so it includes both edges internal to i and edges with exactly one endpoint in i. Operation `areNeighbors` takes as input two community IDs and reports in constant time whether the two communities are connected or not. The constant time factor is due to an underlying `HashMap` offering operation `contains` in time $\mathcal{O}(1)$. Operation `areInSameCommunity` takes as input two vertices and then returns in constant amortized time whether they belong to the same community by calling `find` on each of them and then comparing their community IDs. Operation `communityOf` takes as input a vertex and then reports in constant amortized time the community ID of this vertex by returning the value returned by calling `find` on the vertex. Operation `neighborhood` takes as input a community ID i and returns in time $\mathcal{O}(deg(i)) = \mathcal{O}(n)$ a `TreeSet<Integer>` containing all neighboring communities of i. Operation `merge` takes as input two community IDs i and j and then merges the two communities by calling the `union` op-

eration. Then the array `degrees` is updated in time $\mathcal{O}(1)$. Furthermore, all neighbors of j in `neighbors(j)` are moved to `neighbors(i)` in time $\mathcal{O}(deg(j))$, `neighbors(j)` is cleared in constant time and the number of communities is reduced by one.

Table 7.4: **Metagraph**

Method	Runtime
areNeighbors(int,int): boolean	$\mathcal{O}(1)$
areInSameCommunity(int,int): boolean	$\Theta(1)$ amortized
numCommunities(): int	$\Theta(1)$
communityOf(int): int	$\Theta(1)$ amortized
degree(int): int	$\Theta(1)$
neighborhood(int): TreeSet<Integer>	$\mathcal{O}(deg(i))$
merge(int,int): void	$\mathcal{O}(max\{deg(i), deg(j)\})$

7.1.3 The Algorithm

In this section the pseudo code of the CNM algorithm, as designed by the author of this thesis, on the basis of the CNM article, is presented.

The main procedure of the algorithm is given in Algorithm 4. Initially the data structures are initialized and then a while loop is entered and will continue as long as the max heap H is not empty and the maximum community pair of H will not result in a decrease in modularity Q. At each iteration the maximum element, denoted $(i, j, \Delta Q_{ij})$, of H is removed and the corresponding communities are merged. The amalgamation of the two communities includes updating all of ΔQ, H, M, a, and Q. After the maximum element of H is removed from H in time $\mathcal{O}(\log n)$ and stored in a variable, operation `merge` is called on M, with parameters i and j, taking time $\mathcal{O}(max\{\deg(i), \deg(j)\}) = \mathcal{O}(n)$. Then element $(j, \cdot)$ is removed from both ΔQ^h and ΔQ^t, both operations taking time $\mathcal{O}(\log n)$. Then a set of neighbors of i and j are created in time $\mathcal{O}(k_i + k_j) = \mathcal{O}(n)$ by scanning ΔQ_i^t and ΔQ_j^t. This set is then iterated over and one of the three update rules, mentioned in section 5.3.2, is called, depending on whether the given community is a neighbor of i or j or both of them. Then row j is emptied in both ΔQ^t and ΔQ^h. Then the elements with indices i and j on H are removed and the current maximum element in row i in ΔQ is placed on H. There will no longer be an element with index j on H since this row is emptied in ΔQ. Finally the algorithm outputs the communities after the while loop has ended.

The details of how ΔQ^t and ΔQ^h are updated according to update rule 1 is given in Algorithm 5. We have omitted the code for update rules 2 and 3 since it is almost identical with the exception of how the new value ΔQ_{ik} is computed. These computations were explained in section 5.3.2. For each of the three rules we compute the new value of $\Delta Q_{ik} = \Delta Q_{ki}$ and then we update ΔQ_{ik}^t, ΔQ_{ki}^t, ΔQ_{ik}^h, and ΔQ_{ki}^h with this new value. We also remove the element associated

Algorithm 4: The CNM algorithm

---

input : An input graph G

output: A partition of G into communities.

Initialize ΔQ, H, M, a, and Q

while H *is not empty and* $H.max.modChange > 0$ **do**

    $(i, j, \Delta Q_{ij}) \leftarrow H.deleteMax()$

    // join communities i and j

    $M.merge(i, j)$

    $\Delta Q_i^t.remove(j)$

    $\Delta Q_i^h.remove(j)$

    Let *neighbors* be an ascending list of communities that are neighbors of i or j or both, excluding i and j

    for *every neighbor* $k \in neighbors$ **do**

        if i, j, k *are pairwise connected* **then**

            UPDATEDELTAQCASE1$(\Delta Q, i, j)$

        else if i *and* k *are connected but* j *and* k *are not* **then**

            UPDATEDELTAQCASE2$(\Delta Q, i, j)$

        else if j *and* k *are connected but* i *and* k *are not* **then**

            UPDATEDELTAQCASE3$(\Delta Q, i, j)$

        UPDATEMAXHEAP(H, k)

    $\Delta Q_j^t.clear()$

    $\Delta Q_j^h.clear()$

    $H.delete(j)$

    $H.delete(i)$

    $(maxIndex, maxKey) \leftarrow \Delta Q_i^h.max()$

    $H.insert(i, (i, maxIndex, maxKey))$

    UPDATEA(a, i, j)

    $Q \leftarrow Q + \Delta Q_{ij}$

Output communities

---

with row j from both ΔQ_k^t and ΔQ_k^h since row j no longer is in use because community i has eaten up community j.

Updating ΔQ involves making changes to the rows of community i and every neighboring community k of i or j and thus we may have to update H with regard to both i and k. Updating H with regard to k is shown in Algorithm 6. The code is straightforward in that it compares the current maximum value in row k of ΔQ to the value of the element on H with index k. If the current maximum element in row k of ΔQ is greater than the previously recorded value on H for row k (with index k), then we update H such that the current maximum value in ΔQ_k is bound to index k on H.

Algorithm 5: Update rule 1. The subprocedure for updating ΔQ when communities i and j are being merged.

---

input : ΔQ, three pairwise connected communities i, j, k
output: No output but ΔQ^t and ΔQ^h reflects that i and j are merged into one community i

$newValue \leftarrow \Delta Q_{ik}^t + \Delta Q_{jk}^t$
$\Delta Q_i^t.put(k, newValue)$
$\Delta Q_k^t.put(i, newValue)$
$\Delta Q_k^t.remove(j)$
$\Delta Q_i^h.increaseKey(k, newValue)$
$\Delta Q_k^h.increaseKey(i, newValue)$
$\Delta Q_k^h.remove(j)$

Algorithm 6: The sucprocedure for updating the max heap H, that contains the maximum element from every row of ΔQ.

---

input : Max heap H, integer k representing a community
output: H is updated such that the current maximum element in row k of ΔQ is reflected in H for index k.

$currentMaxIndexInRowK \leftarrow \Delta Q_k^h.maxIndex()$
$currentMaxModChangeInRowK \leftarrow$
$\Delta Q_k^h.keyOf(currentMaxIndexInRowK)$
$currentMaxModChangeFromRowKOnMaxHeap \leftarrow$
$H.keyOf(k).modularityChange()$
if $currentMaxModChangeInRowK >$
$currentMaxModChangeFromRowKOnMaxHeap$ **then**
$\quad\quad H.increaseKey(k, (k, currentMaxIndexInRowK, currentMaxModChangeInRowK))$

7.2 Testing the algorithm

Zachary's karate club

When the algorithm is run on the famous Zachary's karate club network [8], we get a partition into three communities, as shown in Figure 7.1. The results are rather good but we would prefer that vertices $\{25, 26, 29, 32\}$ were included in the community consisting of vertices $\{15, 16, 19, 21, 23, 24, 27, 28, 30, 31, 33, 34\}$. With this single exception we deem this to be a good partition into communities.

Benchmark graphs by Lancichinetti and Fortunato

In this section we will test the algorithm on some benchmark graphs generated by a C++ program and underlying algorithm developed by Lancichinetti and
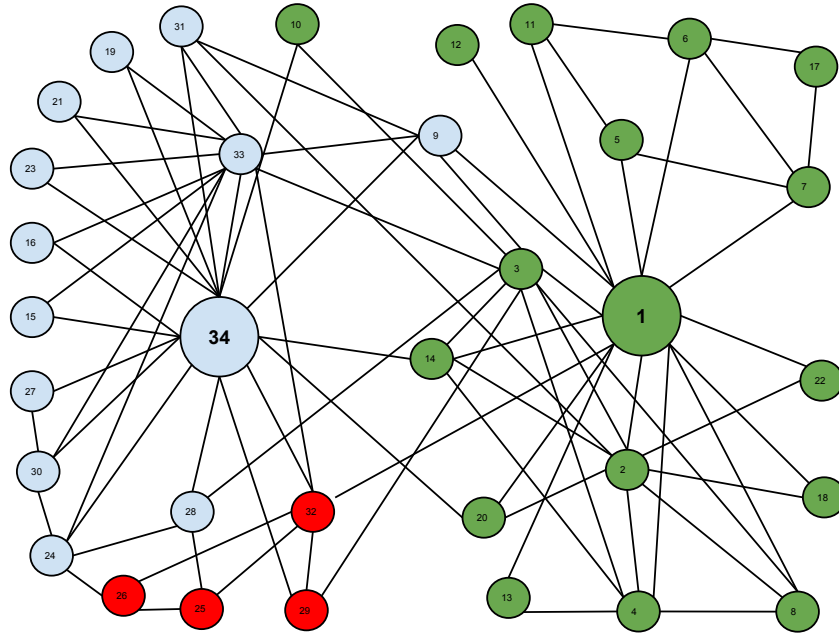
Figure 7.1

Fortunato [17]. The program generates undirected unweighted random graphs with the possibility of overlapping communities based on the following main parameters:

- N : the number of nodes in the graph

- k : the average degree of the graph

- maxk : the maximum degree of the graph

- mu : the topological mixing parameter defined by $k_i^{in} = (1 - mu)k_i$. Here, k_i^{in} is the number of neighbors of vertex i with at least one community in common with i.

- minc : the minimum number of vertices in a community

- maxc : the maximum number of vertices in a community

The output of the program includes an edge list defining the generated graph and a file containing the correct community partition of the graph. The format of this file is two columns of numbers, the first one containing the vertices and the second one their corresponding communities.

The actual testing is done as follows. The implementation of the CNM algorithm by the author of this thesis includes a parser that parses the generated

graph file and constructs the corresponding graph. The algorithm is run on the graph and a two column list is computed showing the vertices in the first column and their corresponding communities in the second column. This two column list is compared to the two column list generated by the program and the number of misplaced vertices is counted.

Results of several tests using these benchmark graphs are given in Table 7.5.

Test #	N	k	maxk	mu	minc	maxc	# misplaced vertices
1	64	16	20	0.1	8	8	48
2	64	16	20	0.1	32	32	0
3	128	16	20	0.1	32	32	32
4	256	16	20	0.1	32	32	3
5	128	16	20	0.1	8	8	88 or 32
6	64	8	20	0.1	8	8	6
7	64	16	20	0.1	16	16	0
8	64	6	10	0.1	4	10	9
9	64	5	12	0.1	4	16	16
10	64	4	14	0.1	4	18	13
11	128	8	20	0.1	8	8	64
12	128	16	20	0.1	16	16	3
13	128	6	10	0.1	4	10	70
14	128	5	12	0.1	4	16	60

Table 7.5: Tests on benchmark graphs by Lancichinetti and Fortunato.

1. In test number 1 the algorithm found two big communities whereas the graph really has eight distinct communities, each with eight vertices. Thus $6 \cdot 8 = 48$ vertices are misplaced. This is an example of the resolution limit problem mentioned in section 4.3.

2. In test number 2 a graph on 64 vertices and all vertices are correctly placed by our algorithm.

3. In test number 3 the result was 32 vertices (all actually belonging to the same community) being absorbed into another community.

4. In test number 4 the result was 3 obviously misplaced vertices, each from a separate community. The most likely explanation to this result is the simplicity of the modularity function causing the algorithm to misplace the vertices.

5. In test number 5 we are generating a graph on 128 vertices where the average degree is 16 and where every community has to consist of 8 vertices. Thus the graph has 16 communities where pairs of them act as larger communities. So we may actually conceive this as a graph consisting of eight communities. The algorithm finds six communities of sizes 16, 16,

16, 19, 30, and 31. Three vertices in the group of 19 vertices are clearly misplaced. The second last group of 30 vertices consists of four communities merged together and where one of them is missing two of its vertices. The last group of 31 vertices consists of four communities merged together and where one of them is missing one of its vertices. In total, 88 vertices are misplaced if we should expect 16 communities, but only 32 vertices if we expect eight communities, each of 16 vertices.

6. In test number 6 a graph on 64 vertices is built. It has four communities of size eight and two communities of size 16. Our CNM algorithm found six communities, of sizes 8, 9, 9, 10, 12, and 16. The first community is perfect, the second and third both include a vertex that should not be included, the foruth community includes two vertices that should not be included, the fifth community misses four vertices that should be included, and the sixth community misses two vertices it should have included and includes two vertices it should not have included.

7. In test number 7 a graph on 64 vertices is built. It has three communities of sizes 16, 16, and 32. The result of our CNM algorithm is totally equal to the answer.

8. In test number 8 a graph on 64 vertices and nine communities of sizes 5, 5, 6, 6, 7, 7, 9, 9, and 10 is built. Our CNM algorithm found eight communities of sizes 5, 6, 7, 7, 8, 9, 10, and 12. Two actual five vertex communities are merged into one community by our algorithm and thus five vertices are misplaced. The 12 vertex community found by our algorithm contains three vertices that it should not contain, and one of the communities on seven vertices contains one vertex it should not contain. In total nine vertices are misplaced.

9. In test number 9 a graph on 64 vertices and seven communities of sizes 10, 6, 8, 6, 11, 13, and 10 is built. Our CNM algorithm found five communities of sizes 10, 11, 21, 11, and 11. The community on ten vertices contains two vertices that it should not contain, the community on eleven vertices is perfect, the community on 21 vertices consists of three communities on 6, 6, and 10 vertices respectively merged together, but one of the vertices in the 10 vertex community is missing. If we count the 10 vertex community as the base community, this community contains 12 vertices it should not contain and misses one vertex that it should contain. In total, 16 vertices are misplaced.

10. In test number 10 a graph on 64 vertices and seven communities of sizes 16, 7, 10, 4, 11, 9, and 7 is built. Our CNM algorithm found five communities with a total of 13 misplaced vertices. The communities on four and nine vertices are merged into one community and one of the vertices in the community on 16 vertices is also included. This accounts for five misplaced vertices. Furthermore, the two communities on seven vertices each are merged into one community, giving seven additional misplaced vertices.

11. In test number 11 a graph on 128 vertices and seven communities of sizes 16, 16, 16, 16, 16, 16, and 32 is built. Our CNM algorithm found three communities of sizes 78, 34, and 16. The community on 16 vertices is perfect. The community on 78 vertices consists of four communities merged together but with one of the communities missing exactly two vertices. The community on 34 vertices consists of two merged communities together with the two missing vertices from the community on 78 vertices. In total 64 vertices are misplaced.

12. In test number 12 a graph on 128 vertices and six communities of sizes 16, 16, 16, 16, 32, and 32 is built. Our CNM algorithm found six communities of sizes 17, 16, 16, 18, 31, and 30. The community on 17 vertices includes a vertex that should be part of the community on 31 vertices, and the community on 18 vertices includes two vertices that should be part of the community on 30 vertices. Thus three vertices are misplaced in total.

13. In test number 13 a graph on 128 vertices and 19 communities of sizes 9, 7, 5, 5, 6, 6, 6, 7, 9, 5, 6, 5, 7, 6, 9, 10, 7, 7, and 6 is built. Our CNM algorithm found eight communities of sizes 12, 35, 14, 12, 20, 20, 9, and 6. The first community on 12 vertices consists of two communities on seven and five vertices merged together. The community on 35 vertices consists of five communities merged together but with two of the communities missing one vertex each. The community on 14 vertices consists of two six vertex communities merged together plus two misplaced vertices from different communities. The second community on 12 vertices consists of two six vertex communities merged together. The first community on 20 vertices consists of three communities merged together plus one misplaced vertex from another community. The second community on 20 vertices consists of three communities merged together but with one of the missing one vertex. The community on nine vertices is perfect, and the community on six vertices is nearly perfect, missing only one of its vertices.

14. In test number 14 a graph on 128 vertices and 15 communities of sizes 9, 14, 8, 9, 9, 5, 6, 6, 7, 6, 12, 8, 12, 8, and 9 is built. Our CNM algorithm found seven communities of sizes 24, 22, 29, 13, 14, 9, and 17. The community on 24 vertices is the amalgamation of two 12 vertex communities. The community on 22 vertices is the amalgamation of three communities of sizes 9, 5, and 7 together with one additional misplaced vertex from another community. The community on 29 vertices is the amalgamation of four communities of sizes 6, 6, 9, and 8 but with the 9 vertex community missing one of the vertices and including one additional vertex from another community. The community on 13 vertices consists of one community missing one of its vertices. The community on 14 vertices consists of two communities of sizes 6 and 8 merged together. The community on nine vertices consists of a nine vertex community lacking one vertex together with one additional misplaced vertex from another community. The community on 17 vertices consists of two communities on eight and

nine vertices merged together. If we, when two or more communities are merged together by our algorithm, regard the largest community as the base community and the vertices from the other communities as misplaced vertices, we get a total of 60 misplaced vertices.

Conclusion

We have learned the following from the previous 14 tests on benchmark graphs.

- The resolution limit problem mentioned in section 4.3 manifests itself in several of the tests such that small communities are merged to form larger communities and such that large communities swallow up small communities.

- The resolution limit problem gives us poor results when we only look at the final numbers of correctly placed vertices. However, as long as for instance two small communities are merged in their entirety into one larger community, the results are not altogether bad. The results are still meaningful and the vertices are still related to each other but we get a more coarse grained view of the communities. A bad result would be that the vertices of a community were swallowed up by several larger communities. We saw some cases where some vertices of a community were spread across several other communities.

- Evaluating the results of running a community detection algorithm on a network is not a straightforward task and giving a single number representing the number of correctly placed vertices is not a fair evaluation of the algorithm. Another aspect is that the results need to be interpreted on the given network before a final judgement is made, since the vertices and edges usually encode real world structural relationships, such as in collaboration networks.

Chapter 8

A divisive algorithm based on the edge clustering coefficient

Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Paris [28] have devised two divisive algorithms, the first one is an improvement of the divisive algorithm of Girvan and Newman [13] (and will henceforth be called the REB algorithm), and the second one is based on the concept of the edge clustering coefficient (and will henceforth be called the RECC algorithm). In this chapter we will present both algorithms, and then we will implement a slightly modified version of the latter algorithm. In chapter 9 we will incorporate the algorithm in a community detection web service.

8.1 The algorithms by Radicchi et al.

8.1.1 The REB algorithm

We first present the REB algorithm that is an improvement of the original algorithm of Girvan and Newman (the GN algorithm), as presented in section 5.2.2.

The problem that Radicchi et al. seek to solve is the fact that the GN algorithm [13] keeps removing edges until none are left and what is left is simply a dendrogram describing the hierarchy of partitions. The algorithm does not say which partition is the best. A couple of years later they introduced an improved version of the algorithm [24] where they use the modularity function, described in chapter 4, to determine the best partition.

Radicchi et al. consider two community definitions in their paper and use them in both the REB algorithm and in the RECC algorithm, discussed in section 8.1.2.

Definition 8.1.1. *Given an input graph $G = (V, E)$, a **strong community** is a set of vertices $W \subseteq V$ with the property that every vertex $v \in W$ has strictly more neighbors in W than in $\overline{W}$, in other words, if $\forall v \in W : k_v^{in} > k_v^{out}$.*

Definition 8.1.2. *Given an input graph $G = (V, E)$, a **weak community** is a set of vertices $W \subseteq V$ with the property that the sum of neighbors inside W is greater than the sum of neighbors in $\overline{W}$, in other words, if $\sum_{v \in W} k_v^{in} > \sum_{v \in W} k_v^{out}$.*

The algorithm can be defined in five steps. In the algorithm a graph structure that is repeatedly changed by edge removals is used and referred to as the workbench graph. Note that this concept is not explicitly mentioned in the paper of Radicchi et al. but it should be in line with their intentions. When the algorithm in steps 4 and 5 tests whether the two connected components satisfy the community definition, the test is done not using the workbench graph but using an adjacency matrix storing the initial structure of the graph.

1. Choose either the weak or strong community definition.

2. Compute the edge betweenness value for every edge in the workbench graph.

3. If there are no more edges left for consideration, then stop the algorithm.

4. Remove the edge $\{u, v\}$ with the highest edge betweenness value from the workbench graph.

5. Recalculate the edge betweenness value for every edge that was affected by the last edge removal.

6. If the connected component of u and v is not split into two connected components, then go to step 3.

7. If the connected component of u and v is split into two connected components G_u and G_v, and if both of them satisfy the community definition, define G_u and G_v as separate subcommunities of the initial community. See Figure 8.1. Go to step 3.

8. If the connected component of u and v is split into two connected components G_u and G_v, and if not both of them satisfy the community definition, then discard this split, put edge $\{u, v\}$ back into the workbench graph, mark it such that it is never considered for removal again, and go to step 3.

The justification of steps 4 and 5 is the following observation. If a network is split randomly into two connected components, where one of them is very large and the other one correspondingly small, the very large connected component is almost always a community. The problem is addressed by observing a random ER graph (see section 2.3) with n vertices. If we split the graph into two parts with αn and $(1 - \alpha)n$ vertices each, for $\alpha > 0.5$, what is then the probability that the largest connected component, containing αn vertices, satisfies one of the community definitions defined above? Raddichi et al. points out that when n is sufficiently large, the probability is very close to a step function around $\alpha = 0.5$,
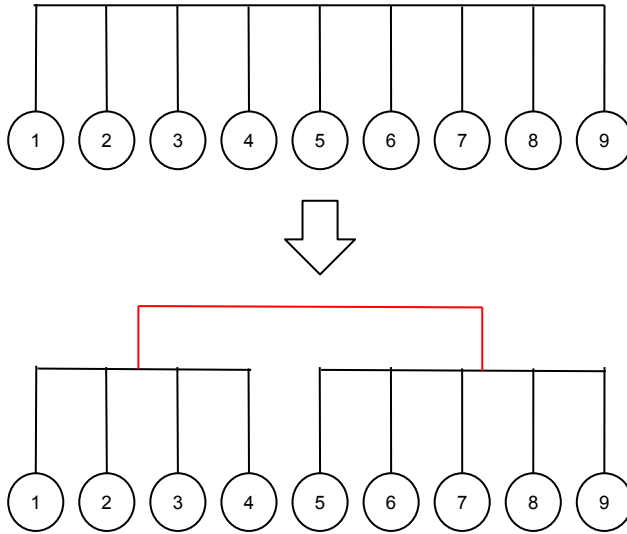
Figure 8.1: An example of step 3 in the algorithm of Radicchi et al. Vertices $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ form a community and is split into two subcommunities, $\{1, 2, 3, 4\}$ and $\{5, 6, 7, 8, 9\}$. The red line represents an edge whose removal will split community $\{1, 2, ..., 9\}$ into communities $\{1, 2, 3, 4\}$ and $\{5, 6, 7, 8, 9\}$.

and hence, when a random ER graph is randomly split into two parts, the largest one is a community with high probability, whereas it is very unlikely that both parts are communities. Therefore, when the algorithm only accepts splits when both the resulting connected components are communities, we are able to correctly identify random graphs. Since they lack community structure the algorithm will not permit the deletion of an edge with the result that we get one connected component that is a community and another connected component that is not a community. Thus when the algorithm is given a random ER graph as input, the result will, with high probability, be exactly one large community comprising all the vertices of the graph.

8.1.2   The RECC algorithm

The second algorithm devised by Radicchi et al. is identical to the divisive algorithm of Girvan and Newman with the exception that it uses the concept of an edge clustering coefficient instead of the edge betweenness concept. Whereas the algorithm of Girvan and Newman at every iteration removes the edge with the highest edge betweenness value, the algorithm of Radicchi et al. removes the edge with the lowest edge clustering coefficient. The edge clustering coefficient of an edge $\{i, j\}$ is given by the equation
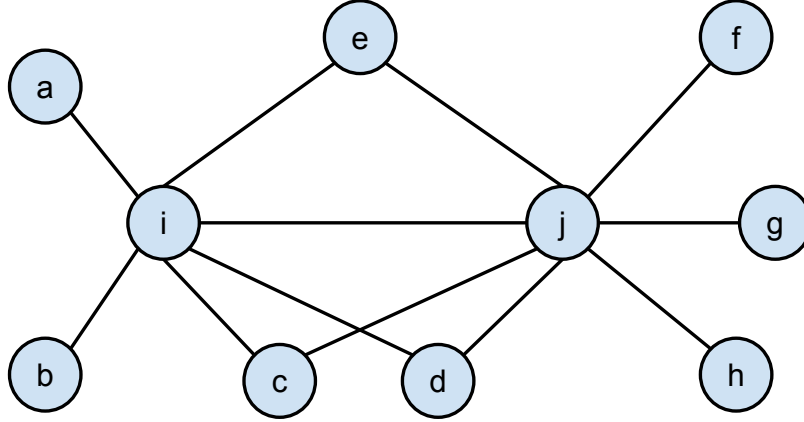
Figure 8.2: The edge clustering coefficient of edge $\{i, j\}$ is $3/5$ since it is part of 3 triangles and the best achievable scenario is that every neighbor of i, except j, that is, 5 neighbors, was a neighbor of j as well.

$$C_{ij}^{(3)} = \frac{z_{ij}^{(3)} + 1}{min\{k_i - 1, k_j - 1\}} \tag{8.1}$$

$z_{ij}^{(3)}$ is the number of triangles that edge $\{i, j\}$ is part of and $min\{k_i - 1, k_j - 1\}$ is the maximum possible number of triangles that edge $\{i, j\}$ can be part of. Adding 1 to the numerator ensures that edges not part of any triangle gets a positive value. Also $C_{ij}^{(3)}$ should be set to $+\infty$ if $min\{k_i - 1, k_j - 1\} = 0$, which happens when at least one of the vertices has only one neighbor. Setting the coefficient to $+\infty$ signals that such an edge should not be removed and the edge is considered last in the algorithm.

Radicchi et al. thus states that $min\{k_i - 1, k_j - 1\}$ is the maximum number of triangles that edge $\{i, j\}$ can be part of. The idea behind this seems to be that the best achievable scenario is that the vertex of minimum degree, say i, shares all his $k_i - 1$ neighbors other than j with j, that is, that $N(i) \setminus \{j\} = N(j) \setminus \{i\}$. See Figure 8.2 for an example.

This definition is used in a divisive algorithm where at every iteration the edge with the lowest edge-clustering coefficient is removed from the graph. The intuition behind this algorithm is that edges connecting different communities are included in few or no triangles, whereas edges inside communities appear in considerably more triangles (see Figure 8.3). Thus $C_{ij}^{(3)}$ is a measure of how highly a given edge is embedded in a community. A high value of $C_{ij}^{(3)}$ indicates that edge $\{i, j\}$ is well embedded in some community, whereas a low value indicates that edge $\{i, j\}$ is an inter-community edge.

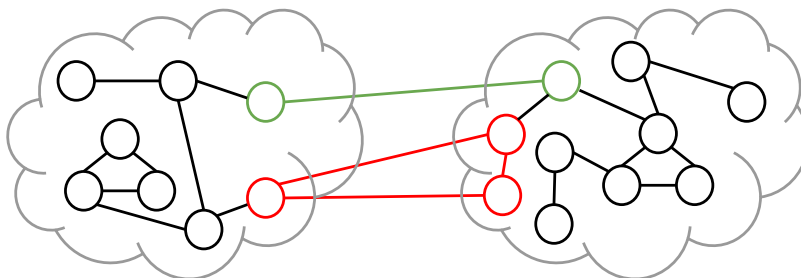A more general definition can also be defined by considering cycles of length g:

Figure 8.3: The idea behind the edge clustering coefficient based algorithm of Radicchi et al. is that edges connecting different communities are included in few or no triangles, whereas edges inside communities appear in considerably more triangles. The red triangle in the figure is something we don't expect to see much of. The green edge, however, is a typical inter-community edge.

$$C_{ij}^{(g)} = \frac{z_{ij}^{(g)} + 1}{s_{ij}^{(g)}} \qquad (8.2)$$

where $z_{ij}^{(g)}$ is the number of cycles of length g edge $\{i, j\}$ is part of, and $s_{ij}^{(g)}$ is the maximum possible number of cycles of length g that edge $\{i, j\}$ can be part of.

For the sake of completeness we first present the new algorithm in prose. The reader should note that this listing is more elaborate than the presentation of the algorithm in the paper of Radicchi et al. The triangle counting is not mentioned as an initial step in the original paper, and neither is any priority queue mentioned in the paper. However, the red line in the presentation below should be in accordance with the paper of Radicchi et al.

1. Choose either the weak or strong community definition.

2. Count the number of triangles that every edge in the graph is part of.

3. Compute the edge clustering coefficient C_e for every edge e and put every pair (e, C_e) into an indexed minimum priority queue Q, ordered by the clustering coefficient C_e.

4. Retrieve and remove the minimum pair (e, C_e), where $e = \{u, v\}$, from Q and then remove e from the workbench graph.

5. **If** the connected component of u and v is split into two connected components G_u and G_v, and if not both of them satisfy the community definition chosen in step 1, then discard this split, put edge $\{u, v\}$ back into the workbench graph, mark it such that it is never again considered for removal, and go to step 8.

6. **Else if** the connected component of u and v is split into two connected components G_u and G_v, and if both of them satisfy the community definition, define G_u and G_v as separate subcommunities of the initial community (see Figure 8.1), and go to step 8.

7. **Else if** the connected component of u and v is not split into two connected components, then go to step 8.

8. If e was removed (and not put back into the workbench graph), then update in Q the pair of every edge that was affected by the removal of e. Go to step 4.

8.1.3 The running time of the RECC algorithm

Radicchi et al. states that the running time of the algorithm is $\mathcal{O}(m^2)$, whereas Fortunato [11] in his review article claims that the running time is $\mathcal{O}(m^4/n^2)$, or $\mathcal{O}(n^2)$ on sparse graphs.

In the following we give an independent analysis of the algorithm where we make several assumptions about how the implementation is done. We will here define a sparse graph to be a graph such that $m < n$. Radicchi et al. do not analyze the triangle counting part, for which we refer the reader to section 8.2.3, but no matter what kind of implementation one chooses, one can not hope to achieve a running time better than $\mathcal{O}(nd_{max}^2)$, where d_{max} is the maximum vertex degree in the graph.

The algorithm handles all m edges in the graph and possibly removes them. Testing whether an edge removal results in its component being split into two connected components can be done with DFS or BFS and this takes $\mathcal{O}(n+m)$ time (and $\mathcal{O}(n)$ if the graph is sparse). If the component is split into two components, then in the worst case both components have to be tested against the community definition. To test a component $\mathcal{C}$ against a community definition we have to loop through every vertex of $\mathcal{C}$ and for every vertex we have to loop through its entire neighborhood to see if each neighbor is inside or outside $\mathcal{C}$. The number of vertices in $\mathcal{C}$ is upper bounded by n and the number of neighbors of a vertex in $\mathcal{C}$ is upper bounded by d_{max}. So testing if $\mathcal{C}$ is a community takes time $\mathcal{O}(nd_{max}) = \mathcal{O}(nm)$ (and time $\mathcal{O}(n^2)$ if the graph is sparse). Thus for the main part of the algorithm, not including the triangle counting, we get the running time $\mathcal{O}(m(n + m + 2nd_{max})) = \mathcal{O}(nmd_{max})$ (and the running time $\mathcal{O}(n^2 d_{max})$ for sparse graphs).

All in all we get the running time $\mathcal{O}(nmd_{max})$, and $\mathcal{O}(n^2 d_{max})$ for sparse graphs.

8.2 Implementing the algorithm

In this section we describe our own implementation of the RECC algorithm.

8.2.1 The Data Structures

In this subsection we briefly present the main data structures that are used in our own implementation of the algorithm by Radicchi et al. that uses the edge clustering coefficient. The algorithm is presented in section 8.2.2.

- **H**. This is a `Graph` that is never altered and that shows how the input graph looks like.

- **G**. This is the workbench graph of type `Graph`. It is modified during the course of the algorithm. Both G and H are stored both as adjacency lists and as edge lists.

- **NT**. This is a `HashMap` storing key-value pairs where the keys are `Edge`s and the values are `Integer`s that represent the number of triangles that the edges is part of.

- **M**. This is a `Metagraph` structure that offer union-find operations and is quite similar to the metagraph described in chapter 7.

- **Q**. This is an indexed minimum priority queue used to store tuples (e, C_e), where e is an edge and C_e is the corresponding edge clustering coefficient. The front element of the queue is always the tuple with the lowest valued edge clustering coefficient C_e.

- **EdgeToId**. This is a `HashMap` mapping `Edge`s to unique `Integer`s and is used to support the *indexed* minimum priority queue Q.

8.2.2 The Algorithm

The pesudo code of our own implementation of the algorithm based on the edge-clustering coefficient is given more formally in Algorithm 7.

The algorithm takes as input a graph H that is stored both as an adjacency list and as an edge list. H will never be altered and will be used when we need to know how the original input graph looks like. Initially a copy is made of H and is later on referred to as G. This is the workbench graph from which we will repeatedly remove edges.

Two things are done initially. First, the number of triangles that every edge is part of in H is computed and the results are stored in a `HashMap` referred to as NT. The edges are the keys and the corresponding numbers of edges the values in NT. See section 8.2.3 for a thorough description of how the triangle counting is done.

Secondly, an indexed minumum priority queue Q is filled with one tuple (e, C_e) for every edge e with its corresponding edge clustering coefficient C_e. Q is ordered by the C_e values such that the minimum element of Q at any time is the tuple (e, C_e) containing the edge with lowest valued edge clustering coefficient in G. Every tuple gets a unique integer id and this information is stored in a `HashMap` called *EdgeToId*. Since Q is indexed and we can retrieve

the unique integer id of every edge from $EdgeToId$ in constant time, we can easily change the edge clustering coefficients stored in Q during the course of the algorithm.

After these two steps have been made, the algorithm enters the main loop, that runs until Q have been emptied for all its elements. At every iteration the minimum element (e, C_e) of Q is retrieved and removed from Q, and then $e = \{i, j\}$ is removed from the workbench graph G. If removing e does not make vertices i and j end up in different components, then we continue with the next iteration. If removing e from G leaves i and j in separate connected components and both components are communities, which we determine from the original input graph H, then we are happy and continue with the next iteration. If on the other hand not both components are communities, then we put e back into G.

If e was removed from G and not replaced, then we update Q and NT correspondingly. The code of this subprocedure is given formally in Algorithm 8. This involves scanning the neighborhood of e that consists of every edge f that together with e form a triangle. The value of f in NT must be decreased by 1 since it after the removal of e is part of one less triangle. Then the edge clustering coefficient of f must be computed again from the current version of the workbench graph G and the tuple of f in Q must be updated correspondingly. Here $EdgeToId$ is used to retrieve the unique integer id of f and then this id is used to get hold of the tuple of f in Q.

The main loop ends when Q is empty and then we use DFS to find the connected components of G. Each connected component is a separate community in H.

8.2.3 Efficient triangle counting

In order to make the algorithm described in the previous subsection as efficient as possible we need to count the number of triangles in the input graph as efficient as possible. To be even more precise, we need to count the number of triangles every edge is part of. In this subsection we first investigate efficient triangle counting algorithms for undirected graphs and then we pick the one that best suits our needs.

Schank and Wagner [29] have described and experimentally tested several triangle counting algorithms, of which we will mention only three. They have measured both the execution times of the algorithms and counted the number of triangle operations done by each algorithm on several both real and artificial networks. The rest of this subsection is based on their article.

We consider an undirected input graph $G = (V, E)$, where $n = |V|$. The degree $d(v) := |\{u \in V : \exists \{v, u\} \in E\}|$ of node v is the number of neighbors $u \in V$ of v. The maximal degree of G is defined as $d_{max} := \max\{d(v) : v \in V\}$. A triangle $\triangle = (V_\triangle, E_\triangle)$ of G is a subgraph of G with $V_\triangle = \{u, v, w\}$ and $E_\triangle = \{\{u, v\}, \{v, w\}, \{w, u\}\}$. The number of triangles in G is denoted by $\delta(G)$.

RadicchiAlgorithm 7:

input : A graph $H = (V, E)$
output: A partition into communities

Let G, a copy of H, be the workbench graph;
NT is a HashMap mapping edges to the number of triangles they are part of;
$NT \leftarrow \text{NUMTRIANGLES}(A, H)$;
Let Q be an indexed minimum priority queue storing tuples (e, C_e), where e is an edge and C_e is the edge clustering coefficient of e, ordered by the values C_e;
for $(e = \{i, j\}) \in H$ **do**
    $c \leftarrow (NT.get(e) + 1)/(\min\{k_i - 1, k_j - 1\})$;
    $Q.put((e, c))$;

while Q *is not empty* **do**
    $((e = \{i, j\}), C_{ij}) \leftarrow Q.deleteMin()$;
    Remove e from G;
    if *areInDifferentComponents(G, i, j)* **then**
        Let G_i and G_j be the connected components of i and j respectively;
        if *not (isCommunity(G$_i$) and isCommunity(G$_j$))* **then**
            Add e back to G

    if *e was really removed* **then**
        $\text{UPDATEDATASTRUCTURES}(Q, NT, e)$;

Compute and output all connected components of G

UpdateDataStructures 8:

input : An indexed minimum priority queue Q, a HashMap NT mapping edges to integers representing the number of triangles the edges are part of, and an edge $e = \{u, v\}$ that has been removed from the workbench graph.
output: No output but changes may have been made to both NT and Q.

NT.put(e,0);
for *every vertex w that together with u and v form a triangle in G* **do**
    NT.put($\{u, w\}$, NT.get($\{u, w\}$)-1);
    idOfUW $\leftarrow$ EdgeToId($\{u, w\}$);
    newCoefficient $\leftarrow \dfrac{NT.get(\{u,w\})+1}{min\{G.numNeighbors(u)-1, G.numNeighbors(w)-1\}}$;
    Q.changeKey(idOfUW, (idOfUW,$\{u, w\}$,newCoefficient));
    NT.put($\{v, w\}$, NT.get($\{v, w\}$)-1);
    idOfVW $\leftarrow$ EdgeToId($\{v, w\}$);
    newCoefficient $\leftarrow \dfrac{NT.get(\{v,w\})+1}{min\{G.numNeighbors(v)-1, G.numNeighbors(w)-1\}}$;
    Q.changeKey(idOfVW, (idOfVW,$\{v, w\}$,newCoefficient));

The simplest triangle counting algorithm is a $\binom{n}{3} = \mathcal{O}(n^3)$ time algorithm that for every subset of three vertices checks if there is an edge between every pair of them.

The *node-iterator* algorithm iterates over all vertices in the graph and for every pair of neighbors checks if there is an edge between them. Thus the running time of this algorithm is

$$\sum_{v \in V} \binom{d(v)}{2} \leq \sum_{v \in V} \binom{d_{max}}{2} = \mathcal{O}(nd_{max}^2). \tag{8.3}$$

The last algorithm is the *edge-iterator*. This algorithm iterates over every edge $\{u, v\}$ of the graph and compares the neighbors of u and v. $\{u, v, w\}$ induce a triangle if and only if w is a neighbor both of u and v. If the neighbor list of u is sorted, then the comparison can be done in time $d(u) + d(v)$. Sorting of all neighbor lists can be done in $\sum_{v \in V} d(v)\log d(v)$ time. If we disregard the sorting part, the running time is given by

$$\sum_{\{u,v\} \in E} d(u) + d(v) = \mathcal{O}(md_{max}) \tag{8.4}$$

where $\mathcal{O}(md_{max})$ is not a very accurate bound. A more accurate bound is given by the following amortized analysis. For every edge $\{u, v\}$ we split the cost $d(u) + d(v)$ into two parts, $d(u)$ and $d(v)$, and assign $d(u)$ to u and $d(v)$ to v. Thus in the outer loop a vertex v is passed $d(v)$ times and the running time can equivalently be expressed as

$$\sum_{v \in V} d(v)^2 \leq \sum_{i=1}^{n} d_{max}^2 = \mathcal{O}(nd_{max}^2) \tag{8.5}$$

and so we can see that the running time of the edge iterator is the same as for the node-iterator.

A result from the experimental analysis of Schank and Wagner is that the edge-iterator *in practice* has the best running time of all the algorithms (including the ones not mentioned here). This is fortunate since we want to know the number of triangles that every edge is part of and this is easy to count with the approach of the edge-iterator.

A more detailed presentation of the edge-iterator algorithm is given in Algorithm 9. The algorithm assumes that the input graph is given both as an unsorted adjacency list and as an edge list. Since the adjacency list is not sorted, the algorithm starts by sorting the neighbor list of every vertex. The number of triangles an edge is part of is stored in a HashMap. The total number of triangles in the graph is also kept track of and the algorithm can easily be changed to return this number instead of the HashMap that it actually returns.

The default load factor of the Java class HashMap is 0.75 and we will use this load factor in our algorithm since it gives a good tradeoff between time and space costs. We are so fortunate that we know the number of key-value pairs, $|E|$, that we will store in the HashMap. By setting the initial capacity of the

HashMap to a number slightly larger than $|E|/0.75$, we ensure that no rehash operations will occur during the course of the algorithm.

NumTriangles 9: NUMTRIANGLES is an edge-iterator that iterates over every edge e of the graph and counts the the number of triangles that e is part of.

---

input : A graph $G = (V, E)$ represented both by an adjacency list A and an edge list EL.

output: A HashMap storing the number of triangles that every edge of the graph is part of.

for $v \in V$ **do**
  Sort $A[v]$;

$totalNumTriangles \leftarrow 0$;
Let NT be a HashMap storing the number of triangles that every edge is part of;
for $\{u, v\} \in EL$ **do**
  $numEdgeTriangles \leftarrow 0$;
  $p_u \leftarrow 0$;
  $p_v \leftarrow 0$;
  while $p_u < A[u].size()$ *or* $p_v < A[v].size()$ **do**
    if $A[u][p_u] = A[v][p_v]$ **then**
      $++ totalNumTriangles$;
      $++ numEdgeTriangles$;
      $++ p_u$;
      $++ p_v$;
    else if $A[u][p_u] < A[v][p_v]$ **then**
      $++ p_u$;
    else if $A[u][p_u] > A[v][p_v]$ **then**
      $++ p_v$;
  $NT.put(\{u, v\}, numEdgeTriangles)$;
Return NT

8.2.4 The running time

Let $H = (V, E)$ be the input graph with $n = |V|$ and $m = |E|$.

The number of triangles is initially counted and stored in the HashMap NT using the edge iterator in time $\mathcal{O}(nd_{max}^2) = \mathcal{O}(n^3)$, where d_{max} is the maximum vertex degree in the input graph.

Then the indexed minimum priority queue Q is filled with one tuple for every edge in H in $\mathcal{O}(m)$ time. Computing the edge clustering coefficient for a single edge takes constant time provided that we have access to NT. Adding a tuple to Q also takes constant time.

The main loop of the algorithm has m iterations. After removing an edge $e = \{u, v\}$ from the workbench graph, a depth-first search (DFS) is initiated from u to check if vertices u and v are still in the same connected component in $\mathcal{O}(n + m)$ time. This reveals the community of u but one additional DFS is needed in order to retrieve the community of v as well.

Then, we test at least one of the components to see if they satisfy the community definition. Testing a community $\mathcal{C}$ for both the weak and strong community definition involves iterating over all $|\mathcal{C}| = \mathcal{O}(n)$ vertices in $\mathcal{C}$ and for every such vertex $v \in \mathcal{C}$ we have to iterate over its entire neighborhood in $k_v = \mathcal{O}(m)$ time. Thus the total time spent is $|\mathcal{C}| \cdot k_{max} = \mathcal{O}(n) \cdot \mathcal{O}(m) = \mathcal{O}(nm)$, where k_{max} is the maximum vertex degree in $\mathcal{C}$.

Therefore, the main loop of the algorithm takes $\mathcal{O}(m(n + m + nm)) = \mathcal{O}(nm^2)$, which is not the same as $\mathcal{O}(m^2)$ found by Radicchi et al.

8.3 Introducing one additional parameter to the Radicchi algorithm

In this thesis we introduce one external parameter, ℓ, to the algorithm by Radicchi et al., namely a lower percentage bound in the range $[0, 1]$ on the size of every community in the graph. This parameter is used both in the weak and strong community definitions as follows. A *strong community* is a set of vertices $W \subseteq V$ with the property that every vertex $v \in W$ has strictly more neighbors in W than in $\overline{W}$, and that at the same time satisfies the property that $|W| \geq n \cdot \ell$. A *weak community* is a set of vertices $W \subseteq V$ with the property that the sum of neighbors inside W is greater than the sum of neighbors in $\overline{W}$, and that at the same time satisfies the property that $|W| \geq n \cdot \ell$.

The motivation behind this additional parameter is that if a user of the algorithm is given a specific network, e.g. a collaboration network in the DBLP database, then he may have a certain idea of how big the communities outputted by the algorithm should be. With this parameter he can make sure this need is met.

The relationship between the new parameter, ℓ, and the dendrogram revealed by the original algorithm, which corresponds to $\ell = 0$, should be noted. Given a connected network as input, the original algorithm reveals a dendrogram of partitions into communities, that is, several layers of community partitions. The very first partition found by the algorithm is simply one community comprising all the vertices of the network. The second partition found by the algorithm is the partition into two communities, the third partition found by the algorithm is the partition into three communities, etc. Whereas the last partition found by the GN algorithm consists of one community for every vertex, the RECC algorithm usually ends up with larger communities due to the restriction set by the weak/strong community definition. However, when we introduce ℓ to the algorithm and set the value to a value greater than 0, then the community partition returned by the algorithm may be one further down in the dendrogram.

See Figure 5.2.

If the user of the algorithm wishes to partition two different networks into exactly k communities each, then he may may need to use different values of ℓ. Thus it is not possible to hard code any universal values of ℓ for such a purpose and thus, given a specific network, a user has to experiment with ℓ in order to find the value that gives the desired result.

This need for *algorithm customization* is some of the motivation for the web service presented in chapter 9 that offers community detection in the DBLP database.

8.4 Introducing weighted networks, weighted community definitions, and a weighted ECC

In chapter 9 we are going to present a web service that offers community detection in the DBLP database. This web service will offer community detection in both unweighted and weighted collaboration networks where the weight of an edge represents the number of publications two authors have in common.

For weighted networks, we will define weighted versions of the weak and strong community definitions. Extending the definitions to the weighted case is straightforward and was already done by Castellano et al. [4] the same year Radicchi et al. [28] published their article.

Definition 8.4.1. *Given an input graph $G = (V, E)$ and a lower bound $\ell \in [0, 1]$, a **strong weighted community** is a set of vertices $W \subseteq V$ such that for every $v \in W$, we have that $\sum_{u \in N(v), u \in W} w(uv) > \sum_{u \in N(v), u \notin W} w(uv)$, and with the additional requirement that $|W| \geq n \cdot \ell$.*

Definition 8.4.2. *Given an input graph $G = (V, E)$ and a lower bound $\ell \in [0, 1]$, a **weak weighted community** is a set of vertices $W \subseteq V$ such that $\sum_{u \in W} \sum_{v \in N(u), v \in W} w(uv) > \sum_{u \in W} \sum_{v \in N(u), v \notin W} w(uv)$, and with the additional requirement that $|W| \geq n \cdot \ell$.*

Castellano et al. [4] has introduced a weighted version of the edge clustering coefficient, that we will make use of in chapter 9 when the algorithm is run on weighted networks.

$$C_{ij}^{(3)(W)} = \frac{z_{ij}^{(3)} \cdot w(\{i,j\}) + 1}{min\{k_i - 1, k_j - 1\}} \tag{8.6}$$

Thus strong edges will tend to have high edge clustering coefficients and will be considered for removal later than edges part of the same number of triangles but of weaker strength.

Counting the number of triangles, $z_{ij}^{(3)}$, that edge $\{i, j\}$ is part of, is done precisely as before.

In addition we will also define a bounded community. This is actually not much of a definition as it only requires that a set of vertices contains at least a

certain percentage of all the vertices in the graph. When this definition is used an edge marked for removal by the edge clustering coefficient will be removed as long as it does not result in a split into two communities where one or both of them is below the specified lower bound. If this definition is used with the lower bound equal to 0, then the output will be $n = |V|$ communities, each containing one of the vertices of the graph. With this definition we get an algorithm that resembles the original GN algorithm more in that it indiscriminately removes edges without considering if resulting components are communities.

Definition 8.4.3. *Given an input graph $G = (V, E)$ and a lower bound $\ell \in [0, 1]$, a **bounded community** is a set of vertices $W \subseteq V$ such that $|W| \geq n \cdot \ell$.*

Experiments show not very surprisingly that this definition does not give good partitions, at least not with small lower bounds. The resulting partitions have low modularity value and the number of inter-community edges exceeds the number of intra-community edges.

8.5 Testing the algorithm

We will not test the algorithm by Radicchi et al. on any benchmark graphs. In chapter 7 we tested the CNM algorithm on some artificially generated networks with little luck, if we are only going to compare the number of correctly placed vertices. We saw several cases where two smaller communities were merged into one larger community, when the answer was two separate communities, but how bad is this? As long as the two communities in their entirety are merged into one community, this is not altogether bad. It is much worse when the vertices of one community is absorbed by several larger communities. Also one needs to evaluate who the nodes represent and what the edges represent in order to say a final word on the quality of the partition.

Radicchi et al. [28] addresses this problem in their article and poses the following questions. "One may directly inspect the dendrogram to answer questions like: Are the communities representative of real collaborations between the corresponding scientists? Do they identify specific research areas? Would a generic scientist agree about his or her belonging to a given community? Obviously, all these questions cannot be answered in a definitive and quantitative way."

Defining "good" and "bad" communities thus has a certain subjective aspect to it. Therefore, in the case of the RECC algorithm, and in response to the challenging questions of Radicchi et al., instead of testing the algorithm against benchmark graphs, we will make a web service where anybody can run the RECC algorithm on the real collaboration networks in the DBLP database, and choose which of the community definitions they want to use and which value they want to use for ℓ, and then they can decide for themselves whether the algorithm gives good results or not.

8.5.1 Running the algorithm on a collaboration network

In this subsection we give an example of the results obtained by running the algorithm with the web service introduced in chapter 9 with the weak community definition and the lower bound $\ell = 0.1$ on the following four collaboration networks of researcher Fredrik Manne:

1. The unweighted network including Manne

2. The unweighted network excluding Manne

3. The weighted network including Manne

4. The weighted network excluding Manne

The results of running the algorithm on the weighted networks include the number of coauthored publications with Manne in parentheses behind each author. These results also include the total internal edge weight inside the community in parentheses behind the community number. In the results of running the algorithm on networks excluding Manne the text CC appears behind some community numbers to indicate that the community is a connected component.

The unweighted network including Manne

The unweighted collaboration network including Manne contains $51 + 1$ authors and 146 coauthor relationships. We obtain a partition into 3 communities with the modularity value 0.3321917808219178, where 100 edges are intra-community edges and the remaining 46 edges are inter-community edges. The communities are as follows.

- **Community 1** (size: 24): Fredrik Manne, Bengt Aspvall, Ferdinando Cicalese, Michelangelo Grigni, Mahantesh Halappanavar, Magnús M. Halldórsson, Johannes Langguth, Phillip Merkey, Rodica Mihai, Morten Mjelde, Randi Moe, Bjørn Olstad, Laurence Pilard, Peter Sanders, Sadia Sharmin, Alicia Thorsen, Sébastien Tixeuil, Bora Uçar, Jianping Wang, Xin Wang 0001, Qin Xin, Xiaolan Yao, Yan Zhang, Zeyu Zheng.

- **Community 2** (size: 23): Ankit Agrawal, Ariful Azad, Rob H. Bisseling, Jean R. S. Blair, Erik G. Boman, Doruk Bozdag, Ümit V. Çatalyürek, Alok N. Choudhary, Pradeep Dubey, Assefaw Hadish Gebremedhin, Salman Habib, Kamer Kaya, Wei-keng Liao, Füsun Özgüner, Diana Palsetia, Md. Mostofa Ali Patwary, Alex Pothen, Peder Refsnes, Nadathur Satish, Tor Sørevik, Narayanan Sundaram, Arijit Tarafdar, Tom Woods.

- **Community 3** (size: 5): Petr A. Golovach, Pinar Heggernes, Pim van 't Hof, Daniël Paulusma, Michal Pilipczuk.

The unweighted network excluding Manne

The unweighted collaboration network excluding Manne contains 51 authors and 95 coauthor relationships. We obtain a partition into 9 communities with the modularity value 0.4105263157894737, where 78 edges are intra-community edges and the remaining 17 edges are inter-community edges. The communities are as follows.

- **Community 1** (size: 18): Ankit Agrawal, Jean R. S. Blair, Erik G. Boman, Alok N. Choudhary, Pradeep Dubey, Assefaw Hadish Gebremedhin, Salman Habib, Wei-keng Liao, Randi Moe, Diana Palsetia, Md. Mostofa Ali Patwary, Alex Pothen, Peder Refsnes, Nadathur Satish, Tor Sørevik, Narayanan Sundaram, Arijit Tarafdar, Tom Woods.

- **Community 2** (size: 2, CC): Bengt Aspvall, Magnús M. Halldórsson.

- **Community 3** (size: 10): Ariful Azad, Rob H. Bisseling, Doruk Bozdag, Ümit V. Çatalyürek, Mahantesh Halappanavar, Kamer Kaya, Johannes Langguth, Füsun Özgüner, Peter Sanders, Bora Uçar.

- **Community 4** (size: 7): Ferdinando Cicalese, Jianping Wang, Xin Wang 0001, Qin Xin, Xiaolan Yao, Yan Zhang, Zeyu Zheng.

- **Community 5** (size: 9): Petr A. Golovach, Pinar Heggernes, Pim van 't Hof, Rodica Mihai, Morten Mjelde, Daniël Paulusma, Laurence Pilard, Michal Pilipczuk, Sébastien Tixeuil.

- **Community 6** (size: 1, CC): Michelangelo Grigni.

- **Community 7** (size: 2, CC): Phillip Merkey, Alicia Thorsen.

- **Community 8** (size: 1, CC): Bjørn Olstad.

- **Community 9** (size: 1, CC): Sadia Sharmin.

The weighted network including Manne

The weighted collaboration network including Manne contains 51 + 1 authors and 146 coauthor relationships. We obtain a partition into 3 communities with the modularity value 0.339041095890411, where 100 edges are intra-community edges and the remaining 46 edges are inter-community edges. The total weight on the intra-community edges is 590 and the total weight on the inter-community edges is 228. The communities are as follows.

- **Community 1** (size: 30, internal edge weight: 140): Fredrik Manne, Bengt Aspvall (2), Doruk Bozdag (3), Ferdinando Cicalese (2), Pradeep Dubey (1), Michelangelo Grigni (1), Salman Habib (1), Magnús M. Halldórsson (2), Phillip Merkey (1), Rodica Mihai (1), Morten Mjelde (7), Randi Moe (1), Bjørn Olstad (1), Füsun Özgüner (1), Daniël Paulusma (2), Laurence Pilard (5), Peder Refsnes (1), Nadathur Satish (1), Sadia Sharmin

69

(1), Narayanan Sundaram (1), Arijit Tarafdar (1), Alicia Thorsen (1), Sébastien Tixeuil (5), Jianping Wang (1), Xin Wang 0001 (1), Tom Woods (1), Qin Xin (7), Xiaolan Yao (1), Yan Zhang (2), Zeyu Zheng (1).

- **Community 2** (size: 17, internal edge weight: 368): Ankit Agrawal (2), Ariful Azad (1), Rob H. Bisseling (1), Erik G. Boman (3), Ümit V. Çatalyürek (3), Alok N. Choudhary (2), Assefaw Hadish Gebremedhin (8), Mahantesh Halappanavar (2), Kamer Kaya (1), Johannes Langguth (4), Wei-keng Liao (2), Diana Palsetia (2), Md. Mostofa Ali Patwary (7), Alex Pothen (2), Peter Sanders (1), Tor Sørevik (3), Bora Uçar (1).

- **Community 3** (size: 5, internal edge weight: 82): Jean R. S. Blair (6), Petr A. Golovach (2), Pinar Heggernes (2), Pim van 't Hof (2), Michal Pilipczuk (2).

The weighted network excluding Manne

The weighted collaboration network excluding Manne contains 51 authors and 95 coauthor relationships. We obtain a partition into 9 communities with the modularity value 0.37894736842105264, where 72 edges are intra-community edges and the remaining 23 edges are inter-community edges. The total weight on the intra-community edges is 457 and the total weight on the inter-community edges is 246. The communities are as follows.

- **Community 1** (size: 11, internal edge weight: 111): Ankit Agrawal (2), Pradeep Dubey (1), Salman Habib (1), Wei-keng Liao (2), Diana Palsetia (2), Md. Mostofa Ali Patwary (7), Alex Pothen (2), Peder Refsnes (1), Nadathur Satish (1), Narayanan Sundaram (1), Arijit Tarafdar (1).

- **Community 2** (size: 2, internal edge weight: 2, CC): Bengt Aspvall (2), Magnús M. Halldórsson (2).

- **Community 3** (size: 11, internal edge weight: 34): Ariful Azad (1), Rob H. Bisseling (1), Erik G. Boman (3), Doruk Bozdag (3), Alok N. Choudhary (2), Assefaw Hadish Gebremedhin (8), Mahantesh Halappanavar (2), Randi Moe (1), Füsun Özgüner (1), Tor Sørevik (3), Tom Woods (1).

- **Community 4** (size: 10, internal edge weight: 212): Jean R. S. Blair (6), Petr A. Golovach (2), Pinar Heggernes (2), Pim van 't Hof (2), Rodica Mihai (1), Morten Mjelde (7), Daniël Paulusma (2), Laurence Pilard (5), Michal Pilipczuk (2), Sébastien Tixeuil (5).

- **Community 5** (size: 5, internal edge weight: 73): Ümit V. Çatalyürek (3), Kamer Kaya (1), Johannes Langguth (4), Peter Sanders (1), Bora Uçar (1).

- **Community 6** (size: 7, internal edge weight: 24): Ferdinando Cicalese (2), Jianping Wang (1), Xin Wang 0001 (1), Qin Xin (7), Xiaolan Yao (1), Yan Zhang (2), Zeyu Zheng (1).

- **Community 7** (size: 1, internal edge weight: 0, CC): Michelangelo Grigni (1).

- **Community 8** (size: 2, internal edge weight: 1, CC): Phillip Merkey (1), Alicia Thorsen (1).

- **Community 9** (size: 1, internal edge weight: 0, CC): Bjørn Olstad (1).

Conclusion

The algorithm seems to give meaningful communities in both the weighted and unweighted case, and both when including Manne in the network and when excluding him from the network. Excluding Manne from the network help us easily detect the most obvious communities. When it comes to the modularity values we recall from section 4.3 that the maximum modularity value of a given network is dependent both on the size of the graph and on the number of well defined communities and is not necessarily as high as 1. Therefore, it is not possible to compare the modularity value of a network excluding Manne with the modularity value of a network including Manne. We note, however, that the unweighted and weighted networks including Manne have the modularity values 0.3321917808219178 and 0.339041095890411, respectively. We also note that the unweighted and weighted networks excluding Manne have the modularity values 0.4105263157894737 and 0.37894736842105264, respectively. When we compare the number of intra-community edges with the number of inter-community edges, the results seem reasonable as well, since in every case the number of intra-community edges is higher than the number of inter-community edges.

Fredrik Manne gave us the following evaluation of the results: "Out of the four suggestions I prefer the solutions excluding myself, mainly because they partition the list into more clusters. Comparing the weighted and unweighted solutions I believe the weighted one gives most meaning. There are two changes I would have made, "Alex Pothen" belongs in Community 3, and "Randi Moe" and "Tor Sørevik" could have been a group by themselves."

Chapter 9

A community detection web service

In this chapter we present a community detection web service offering community detection in the DBLP database (available at the URL http://dblp.uni-trier.de/) using our implementation of the RECC algorithm presented in chapter 8. DBLP is a service providing open bibliographic information on major computer science journals and proceedings.

The web service is created with the Spring Framework, an open source application framework for the Java platform. A nice and authoritative introduction to the Spring Framework can be found in *Spring in Action*, fourth edition, by Craig Walls [36], senior engineer with Pivotal Software, the developer of the Spring Framework.

The motivation for this web service is twofold. Firstly, as Radicchi et al. [28] points out in their article, the quality of a partition of a network into communities can not be answered in a definitive and quantitative way and one has to evaluate the results on each network in order to say if they are reasonable or not. Also, it is a matter of taste how large communities one wants in the results. Secondly, an algorithm such as the RECC algorithm, where one can choose between community definitions, and choose a lower bound on the size of every community, lends itself to be customized by the user. Each community definition may give meaningful results and they do not necessarily exclude each other. The combination of these two motivating factors have spawned the web service presented in this chapter.

9.1   An overview of the software used

The web service is simply called **DBLP Communities** and is published on the address www.dblpcommunities.com on a GlassFish Server Open Source Edition 4.1 application server. The service is written in Java 8 using version 4.0.3 of the Spring Web MVC Framework. Thus the web service applies the MVC pattern

| URL	Returned
http://dblp.uni-trier.de/pers/xc/ d/Doe:John.xml	The coauthor list of John Doe, containing the name and URL pointer of every coauthor of John Doe and the number of articles they have in common.
http://dblp.uni-trier.de/search/author/api? q=John+Doe&h=1000&c=0 &rd=1a&format=xml	List of hits on authors when searching for John Doe, including names and URL pointers.

Table 9.1: The subset of the XML API of the DBLP database used by our web service.

but also makes use of a service layer that handles the business logic and keeps the controllers nice and clean. The service utilizes Apache Maven as build automation tool, Hibernate Validator 4.0.3.Final for input validation, Jackson Mapper ASL 1.9.5 for mapping POJOs to the JSON format, and a SAXParser for parsing XML documents. The view of the web service utilizes JavaServer Pages (JSP) on the server side. On the client side, the Bootstrap framework is used to give the web service a stylish look that easily adapts its view to both desktop computers, pads, and smart phones, jQuery is used to give user-friendly features to the view, the JavaScript display engine MathJax is used to display mathematical notation, and the JavaScript graph visualization library vis.js is used to draw the collaboration networks.

9.2 The DBLP XML API

Communication with the DBLP server is done through its XML API, which is summarized in Table 9.1. A ceoncept that is much used in the DBLP database is a URL pointer, or `urlpt` as it is actually called, which is a string that uniquely identifies an author in the DBLP database. For John Doe it would be `d/Doe:John`.

The API consists of a set of GET requests for different URL resources that results in the server returning certain XML files. Only two kinds of requests are made use of by DBLP Communities. When a GET request with the URL http://dblp.uni-trier.de/pers/xc/d/Doe:John.xml is sent to the DBLP server, it responds with an XML file containing the coauthor list of John Doe, including both the name and URL pointer of every coauthor of John Doe, and also the number of publications he or she has written with John Doe. A GET request for the resource http://dblp.uni-trier.de/search/author/api? q=John+Doe&h=1000&c=0 &rd=1a&format=xml results in an XML file containing all hits on authors when searching for John Doe.
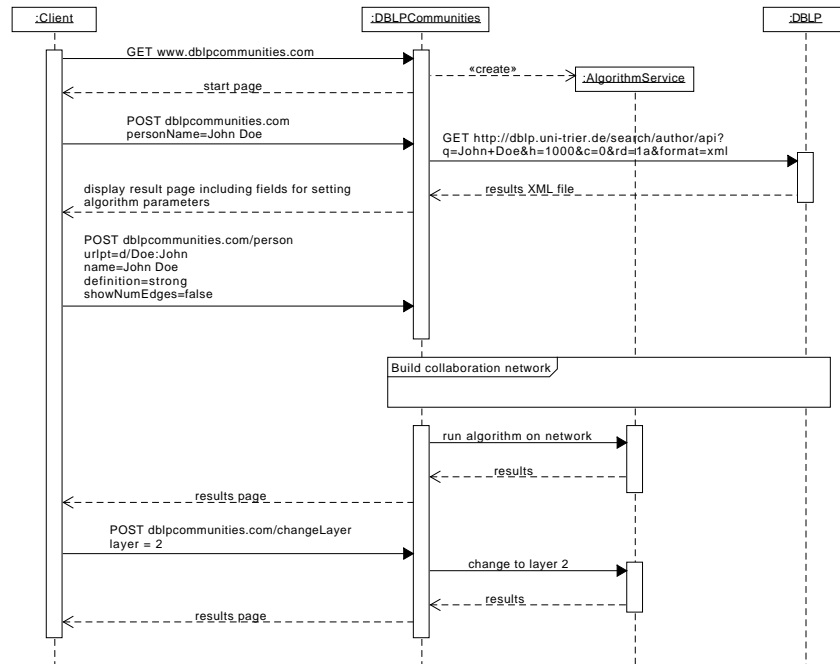
Figure 9.1: A sequence diagram showing typical non-erroneous client interaction with www.dblpcommunities.com.

9.3 How the web service works

Typical non-erroneous client interaction with www.dblpcommunities.com is sketched in the sequence diagrams in Figure 9.1 and Figure 9.3. Note that these diagrams are highly simplified and are only intended to show the most important stuff that is going on in a typical interaction with the web service. Five participants are shown in the diagrams:

- **Client**. A browser guided by a human user.

- **DBLP**. The DBLP server

- **DBLPCommunities**. Our web service that utilizes many different classes to get its work done.

- **AlgorithmService**. One of the many classes used by DBLPCommunities.

- **Graph**. A graph class.

The typical non-erroneous interaction starts when the client sends a GET request to www.dblpcommunities.com. The web service then returns the start

DBLP Communities

Beta version

DBLP Communities offers community detection in the DBLP database.

Introduction

This web page offers community detection in the DBLP database using a divisive hierarchical algorithm based on the following article ...

Show introduction   Hide introduction

Step 1: Find a person whose collaboration network you will do community detection on

Search for a person in the DBLP database

Search

The data provided on the DBLP webpage are released under the Open Data Commons Attribution License (ODC-BY 1.0).

Step 2: Choose algorithm settings

Choose a person	Jan Arne Telle
Graph type	Weighted graph
Community definition	Weak community
Lower bound on the size of every community	10 %
Show detailed information on inter-community edges	☐
Include main author in network	☐

Detect

(a) The front page of www.dblpcommunities.com offering user to search for an author in the DBLP database.

(b) The page offering the user to set various the algorithm parameters.

Figure 9.2

page, shown in Figure 9.2(a), offering the user to search for an author in the DBLP database. The user enters "John Doe" in the search field on the start page and presses the search button. The client then sends a POST request to www.dblpcommunities.com with the parameter `personName` set to "John Doe". Then DBLP Communities sends a GET request to the DBLP server for the resource http://dblp.uni-trier.de/search/author/api? q=John+Doe &h=1000 &c=0 &rd=1a &format=xml, whereupon the DBLP server responds with an XML document containing the hits. DBLP Communities then returns a new page where the user can set various algorithm settings, shown in Figure 9.2(b). The hits are shown in a drop-down list, together with a drop-down list where the user can choose one of the community definitions, a drop-down list where one can specify whether to use a weighted or unweighted network, a drop-down list where the lower bound can be set, a checkbox that can be marked if detailed inter-community information is to be shown on the result page, and a checkbox that can be marked if the main author is to be included in the network. When the user has made his choices and presses a "Detect" button, a POST request is sent to DBLP Communities that includes parameters called `urlpt` and `name`. The corresponding values were set by the choice of element in the person drop-down list and the values were injected by DBLP Communities when the page was sent to the client. The request also includes parameters `definition`, `showNumEdges`, and `includeMainAuthor` – containing the values of the remaining algorithm settings explained above.

9.3.1   Building the collaboration network

DBLP Communities then starts a dialogue with the DBLP server in order to build the collaboration network of John Doe. This part is shown in the sequence diagram in Figure 9.3. First the coauthor list of John Doe is requested in a GET request for the resource http://dblp.uni-trier.de/pers/xc/d/Doe:John.xml. The DBLP server responds with an XML file containing a list of all coauthors of John
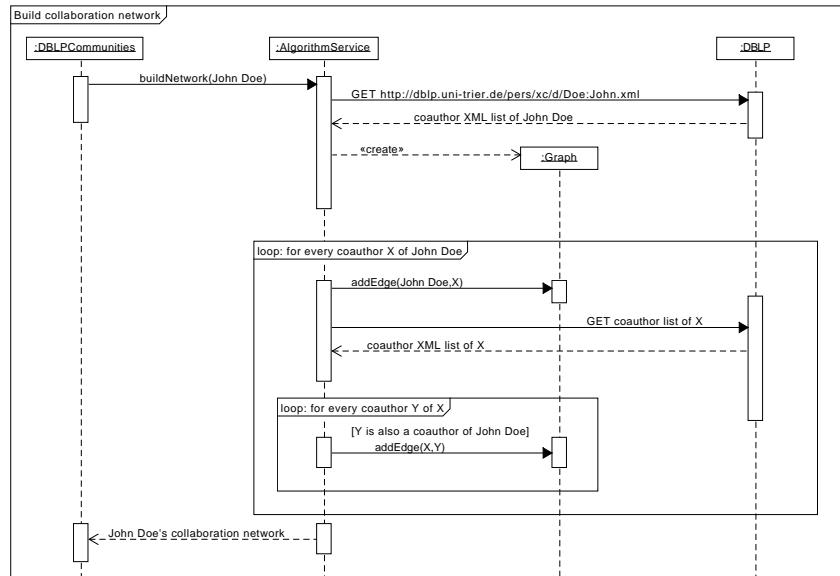
Figure 9.3: A sequence diagram showing how the collaboration network of author John Doe is built.

Doe, including numbers indicating how many publications John Doe has coauthored with each of them. DBLP Communities then downloads the coauthor list of every coauthor of John Doe. Next four different collaboration networks are built:

- An unweighted collaboration network without John Doe

- An unweighted collaboration network with John Doe

- A weighted collaboration network without John Doe

- A weighted collaboration network with John Doe

Every coauthor of John Doe is included as a node in each of the four networks. If two coauthors of John Doe have published together, then an edge is created between them. In the networks containing John Doe, an edge is created between John Doe and every coauthor of his. All four networks are stored in the session for quick access if the user wishes to run the algorithm again on any of them.

9.3.2 The results page

An example of a results page is shown in Figure 9.4. The results page includes an information pane with the following information:

- Whose collaboration network this is

- The definition that was used

- The lower bound that was used

- Whether John Doe is included in the network or not

- The number of communities found

- The number of coauthors of John Doe

- The modularity of the obtained community partition

- The total number of edges in the collaboration network

- The total number of intra-community edges

- The total number of inter-community edges

The results page also contains a pane from which the user can run the algorithm again on the same author but with different settings. When the algorithm is run again with different settings the collaboration network is retrieved from the session and is not rebuilt. The page also contains a pane from which the user can choose a specific layer from the partition hierarchy. Recall that the RECC algorithm is a hierarchical algorithm that starts out with all vertices in one community, then finds a partition into two communities, then into three communities, etc. The whole partition hierarchy, that is, the dendrogram found by the algorithm, is stored in session and can be retrieved by the server in constant time.

At the bottom of the result page the collaboration network is drawn using the JavaScript graph visualization library vis.js, which retrieves the graph in JSON format via a GET request for the resource www.dblpcommunities.com/json/John Doe. A JSON representation of the graph of John Doe is then returned, given that John Doe was the last author the user did community detection on. If this is not the case, then an error message is returned in JSON format. Thus DBLP Communities does not support the creation of arbitrary collaboration networks in JSON format at arbitrary times. The actual layout of the network is done by a graph drawing algorithm implemented in the vis.js library and has nothing to do with the community detection algorithm by Radicchi et al.

9.4   A drawback and its solution

A drawback of DBLP Communities is the way it builds the collaboration networks. The DBLP server limits the number of requests that can be made to it during a certain amount of time. A consequence of this is that the collaboration network of authors with large coauthor lists can not be built. The current version of DBLP Communities notifies the user about this drawback.

A solution to this problem is already created and can be put into production but needs some funding since you can't have your database on a server for free. The solution is described in the following and sketched in Figure 9.5.

A stand-alone Spring Boot application downloads the XML file containing all the publications in the DBLP database. The application goes through every publication in the XML file and creates a node for every author in memory. For every pair of coauthors an edge is made, which is simply a reference from one author object to another author object. As soon as 500 or more authors have been stored in memory a transaction is made to the database with these authors. Authors and coauthor relationships that are not already present in the database are added and the rest is discarded.

The application utilizes Spring Data Neo4j to create a Neo4j database containing one node per author in the DBLP database and with an edge labeled `is coauthor of` between two nodes if their corresponding authors have published together. The database model is shown in Figure 9.6. The resulting database takes almost 430 MB of storage. The application can be put on a server and included in a task scheduler to be run once a month since the XML file of the DBLP database is updated once a month.

The Neo4j database will run in server mode on e.g. GrapheneDB (a service that hosts Neo4j databases for 50 dollars a month if you need 1 GB of storage) and be available through the standard Neo4j server mode REST API. The Neo4j server includes an unmanaged server extension that extends the standard REST API of Neo4j with an additional GET request on the form addressOf-Neo4jDatabase/unmanaged/neighborhood/John Doe. The server responds with the collaboration network of John Doe in JSON format. DBLP Communities, on the other hand, consumes this resource with the help of Spring Data Rest, which automatically transforms the JSON content to a predesigned POJO.

If John Doe has k coauthors, then $k + 1$ GET requests for XML files to the DBLP server are replaced by one GET request to the Neo4j server for the neighborhood of John Doe. The Neo4j database already stores the entire collaboration network of the DBLP database and simply has to return the induced subgraph containing the neighborhood of John Doe, so a drastic speedup can be achieved for authors with many coauthors.

It should, however, be noted that a Neo4j database supporting weighted networks has not been created due to lack of time.

(a) The top of the results page.

(b) The middle of the results page.

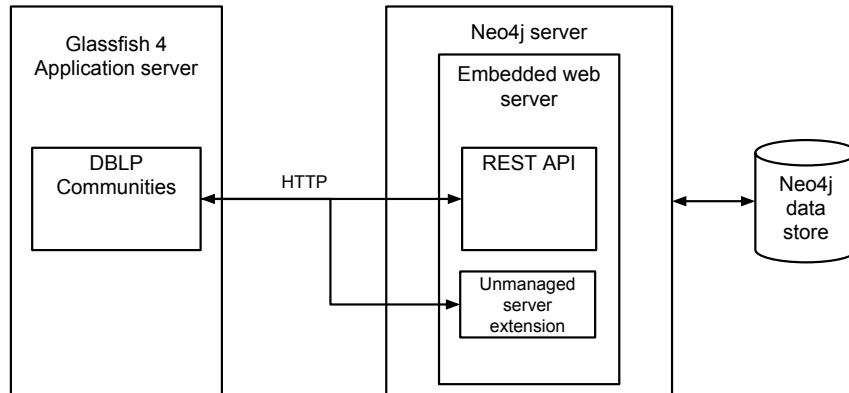(c) The bottom of the results page.

Figure 9.4

Figure 9.5: The alternative application setup. DBLP Communities is backed by a Neo4j database running on a separate server. The REST API is extended with an unmanaged server extension to provide JSON export of collaboration networks.
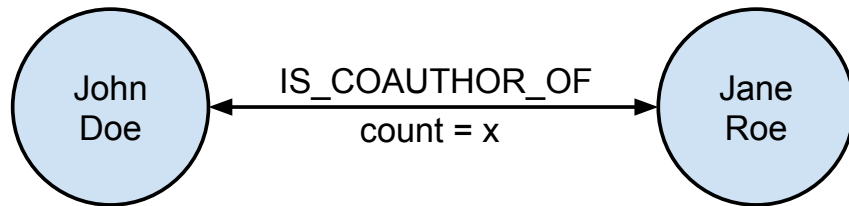

Figure 9.6: The Neo4j database model. John Doe and Jane Roe are two fictional authors in the DBLP database and they have published x articles together. The count attribute is not implemented in the database due to lack of time.

Chapter 10

Conclusion and further research

10.1 Summary

In this thesis, implementations of the CNM and RECC algorithms have been presented together with a web service offering community detection in the DBLP database using the the RECC algorithm and giving the user the opportunity to set various parameters in this algorithm. Testing community detection algorithms is not altogether easy, as was shown in the chapter on the CNM algorithm, as social networks often are hierarchically organized with smaller communities living inside larger communities, and people having different opinions on how large the outputted communities should be. This was some of the motivation for making a web service offering to reveal the different partitions in the dendrogram found by the algorithm and offering to give as output communities of a certain size.

Both the modularity function used by the CNM algorithm and the edge clustering coefficient used by the RECC algorithm are simple functions and are therefore not able to help find the very best partitions into communities. However, they are able to find relatively good partitions and for many purposes they may be good enough. In the end it will be up to the knowledgeable user to evaluate the results on each network, and our web service may help achieve this.

10.2 Further research and application improvements

This sections presents one possible improvement of the RECC algorithm and two ideas on improvements of the web service presented in this thesis.

10.2.1   Improving the RECC algorithm

Radicchi et al. [28] presented results both using the edge clustering coefficient counting triangles, $C_{ij}^{(3)}$, and the edge clustering coefficient counting length four cycles, $C_{ij}^{(4)}$. In our thesis we only implemented the triangle version. Due to the simplicity of the ECC the overall objective of a continued research should be to make a better and more complex coefficient or function that in a better way detects community structures. We pose two questions that should be further investigated.

- Can qualitatively better results be obtained by using a combination of the length three and length four edge clustering coefficients?

- Given a weighted network, can a qualitatively better weighted edge clustering coefficient for an edge $\{i, j\}$ be defined that takes into consideration not only the weight of $\{i, j\}$ but also the weight of every edge of every triangle that $\{i, j\}$ is part of? The rationale for this suggestion is that if i and j both have strong relationships with a third vertex k, then this should perhaps increase the chances of i and j ending up in the same community, even though they are not strongly related themselves.

10.2.2   Improving DBLP Communities

Many improvements can be made to DBLP Communities. However, we think that the following three improvements are the most important.

- A database supporting community detection in both unweighted and weighted collaboration networks in the DBLP database.

- An API offering researchers to easily implement and incorporate their algorithms into DBLP Communities. Thus DBLP Communities could be made into a collaborative testing ground for the community detection community.

- A rating mechanism that harvests user ratings on the combination of algorithm, settings, and collaboration network. Thus one can get a clearer picture on which algorithms and which settings that achieves the best results on each network.

Appendices

Appendix A

Graph theory

In this chapter we present the very basic graph theory – nothing more than is needed to keep pace with the flow of this master thesis.

A.1 Basic terminology

A graph is a pair $G = (V, E)$ of sets, where V is a finite set of vertices (or nodes) and $E \subseteq \binom{V}{2}$ is a collection of edges. Thus, E is a collection of 2-element subsets of V. The edge between vertices 1 and 2 is denoted $\{1, 2\}$ and is equivalent to $\{2, 1\}$. Note that E being a collection allows multiple edges between a fixed pair of vertices. Vertices are usually drawn as dots and edges as lines between two of these dots if the corresponding vertices form an edge, see Figure A.1. If an edge e joins two vertices x and y, then x and y are said to be *adjacent* and e is *incident* with both x and y. An edge joining a vertex x to itself is called a *loop*. A graph without loops or multiple edges is called a *simple* graph. The number of edges incident with a vertex v is called the *degree* of v and is denoted $d(v)$. A vertex of degree 1 is called a *leaf*.
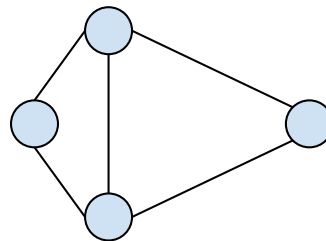

Figure A.1: An example of a graph.

A simple graph with n vertices and where every pair of vertices is connected by a single edge is called a *complete graph* and is denoted by K_n. K_3 is the complete graph on three vertices and is sometimes called a *triangle*.

If $G = (V, E)$ and $G' = (V', E')$ are two graphs such that $V' \subseteq V$ and $E' \subseteq E$, then G' is a *subgraph* of G and we write this as $G' \subseteq G$. If $G' \subseteq G$ and G' contains every edge $uv \in E$ where $u, v \in V'$, then G' is an *induced subgraph* of G.
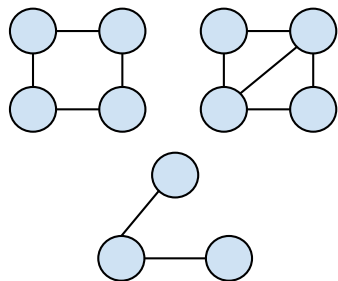
Figure A.2: A graph with three connected components.

A sequence of edges in a graph G of the form $v_0v_1, v_1v_2, v_2v_3, ..., v_{n-1}v_n$ or equivalently $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_n$ is called a *walk* in G. v_0 is the initial vertex of the walk, v_n is the final vertex of the walk, and n is the *length* of the walk. A walk where no two edges are equal is called a *trail*. A trail where no two vertices are equal, except possibly $v_0 = v_n$, is called a *path*. A path where $v_0 = v_n$ is called a *cycle*. The *shortest path* between two vertices u and v is the minimum length path between u and v. The length of such a path is called the *geodesic distance* between u and v. When computing the shortest path between every pair of vertices, the *diameter* of the graph is the maximum length shortest path. A graph G is *connected* if there is a path between every pair of vertices in G. A graph that is not connected consists of a number of connected components, see Figure A.2.

Two vertices are *neighbors* if they are connected by an edge. The *neighborhood* of a vertex v is denoted $N(v)$ and is the set of every neighbor of v.

Bibliography

[1] A Arenas, A Fernández, S Fortunato, and S Gómez. Motif-based communities in complex networks. *Journal of Physics A: Mathematical and Theoretical*, 41(22):224001, 2008.

[2] Stephen P. Borgatti, Martin G. Everett, and Paul R. Shirey. {LS} sets, lambda sets and other cohesive subsets. *Social Networks*, 12(4):337 − 357, 1990.

[3] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 2001.

[4] C. Castellano, F. Cecconi, V. Loreto, D. Parisi, and F. Radicchi. Self-contained algorithms to detect communities in networks. *The European Physical Journal B - Condensed Matter and Complex Systems*, 38(2):311–319, 2004.

[5] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004.

[6] Reuven Cohen and Shlomo Havlin. *Complex Networks: structure, robustness, and function*. Cambridge University Press, New York, 2010.

[7] Leon Danon, Albert Díaz-Guilera, and Alex Arenas. The effect of size heterogeneity on community identification in complex networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2006(11):P11010, 2006.

[8] David Easley and Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, New York, 1st edition, 2010.

[9] Erlend Eindride Fasmer. Cnm on github. `https://github.com/erlfas/CNM`.

[10] Erlend Eindride Fasmer. Dblp communities on github. `https://github.com/erlfas/DBLPCommunities`.

[11] Santo Fortunato. Community detection in graphs. 2010.

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

[13] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Physical Sciences - Applied Mathematics*, 2002.

[14] Benjamin H. Good, Yves-Alexandre de Montjoye, and Aaron Clauset. Performance of modularity maximization in practical contexts. *Phys. Rev. E*, 81:046106, Apr 2010.

[15] Roger Guimerà, Marta Sales-Pardo, and Lu´A. Nunes Amaral. Modularity from fluctuations in random graphs and complex networks. *Phys. Rev. E*, 70:025101, Aug 2004.

[16] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 1970.

[17] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, 78:046110, Oct 2008.

[18] F. Luccio and M. Sami. On the decomposition of networks in minimally interconnected subnetworks. *Circuit Theory, IEEE Transactions on*, 16(2):184–188, May 1969.

[19] Duncan Luce. Connectivity and generalized cliques in sociometric group structure. 1950.

[20] R. Duncan Luce and Albert D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[21] H. Matsuda, T. Ishihara, and A. Hashimoto. Classifying molecular sequences using a linkage graph with their pairwise similarities. *Theoretical Computer Science*, 1999.

[22] Robert J. Mokken. Cliques, clubs and clans. *Quality and Quantity*, 13(2):161 – 173, 1979.

[23] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Phys. Rev. E*, 69:066133, Jun 2004.

[24] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, February 2004.

[25] M.E.J. Newman. Detecting community structure in networks. *The European Physical Journal*, 2004.

[26] Vreda Pieterse and Paul E. Black. Sparse graph — dictionary of algorithms and data structures, August 2008.

[27] M. A. Porter. Small-world network. 7(2):1739, 2012. revision 142132.

[28] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101(9):2658–2663, 2004.

[29] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, WEA'05, pages 606–609, Berlin, Heidelberg, 2005. Springer-Verlag.

[30] Philipp Schuetz and Amedeo Caflisch. Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement. *Phys. Rev. E*, 77:046112, Apr 2008.

[31] Robert Sedgewick and Kevin Wayne. `http://algs4.cs.princeton.edu/24pq/IndexMinPQ.java.html`. [Online; accessed April 2015].

[32] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269 – 287, 1983.

[33] Stephen B. Seidman and Brian L. Foster. A graph-theoretic generalization of the clique concept. *The Journal of Mathematical Sociology*, 6(1), 1978.

[34] Stanford.edu. Single-link and complete-link clustering, 2015.

[35] Ken Wakita and Toshiyuki Tsurumi. Finding community structure in mega-scale social networks: [extended abstract]. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 1275–1276, New York, NY, USA, 2007. ACM.

[36] Craig Walls. *Spring in Action*. Manning Publications Co, Shelter Island, 4th edition, 2014.

[37] Stanley Wasserman and Katherine Faust. *Social Network Analysis, Methods and Applications*. Cambridge University Press, 1994.

[38] Wikipedia. Complete-linkage clustering, 2015.

[39] Wikipedia. Single-linkage clustering, 2015.