University of Bergen

Department of Informatics

Algorithms

Community Detection in Complex Networks

Student:
Herman Møyner Lund

Supervisor:
Professor Fredrik Manne

Master Thesis
June 2017

Acknowledgment

I would first like to thank my thesis supervisor Professor Fredrik Manne, for all the support and help during the work with my thesis. I would also like to thank the algorithmic group, for welcoming me into the group. Finally, I would like to thank my fellow master students for all the support during late nights at the reading hall.

Contents

Chapter 1

Introduction

If we observe a room full of people we expect that many of them will have some kind of relationship to each other. Some may be friends, colleagues, or they might share some mutual interest. Imagine that we want to divide these people into groups based on their relationships. The room could be a class room with students taking the same course. We now want to split the class into work groups, so that groups of friends end up in the same work group, and friends get to work with each other. To accomplish this we need to identify groups of friends. To be able to identify the groups we have asked each student which other students he or she would classify as a friend. One could hope that this would lead to some clearly divided groups of friends, but what if it turns out that students are not only friends with a close little group? Some students have many friends, perhaps the entire class. Other might only know one or two. Perhaps some would classify other students as friends, and it turns out that the friendship is not returned. The goal is to find a grouping such that the students feel like they belong to the group they are assigned to.

To solve this problem, we could use community detection. The goal of community detection is very broadly defined as dividing data into natural groups. In this example the data is students, and natural groups are groups of friends. A classroom is a very small example, and a professor could probably solve this problem without using too much time. Imagine instead that we want to do the same for an entire university, then this becomes an infeasible task to do manually for just one person.

Community detection goes much further than just finding groups of friends. We can try to find communities in any data. Ideally community detection is also able to detect if there does not exist any communities if that is the case. Research on community detection has recently become quite popular, and has been performed on all sorts of data. Communication-networks [5], social

networks [33], and human brain functional networks [26], have all been studied. There has even been research on how community detection could be useful in counter-terrorism [20]. One obvious application from an economical viewpoint is to sort customers into groups based on their interests, to be able to give targeted advertisements based on what other customers in the same group have bought [27].

In this thesis we study the inner workings of one of the most used community detection algorithms, the Louvain-method, a community detection algorithm, developed by Blondel et al. [5]. We take a deeper look at its running-time, and how it groups data. We introduce several variants of the Louvain method, and evaluate how these variations affects both running-time and grouping of data. In addition, we test our variations on several data sets, both from the real world, and on artificial networks.

Our main results are that the order in which the Louvain-method process data does matter. Some of our variants perform better than the original Louvain-method. We also show that by adding thresholds to the Louvain-method, we drastically reduced the running-time.

In Chapter 2, we formally introduce community detection. In Chapter 3 we give a brief overview of community detection algorithms, and their common features. We continue in Chapter 4 with a presentation of modularity, the key building block in how the Louvain-method detects communities. In Chapter 5 we introduce the Louvain-method itself. In Chapter 6 we take a deeper look at the algorithm. Chapter 7 and 8 focuses on alternative ways to process the data sets, and how this affects running-time and grouping. In Chapter 9, we introduce a threshold to the Louvain-method and evaluate the effects of this. We combine the results from 8 and 9 in Chapter 10. In Chapter 11, we test our variants on synthetic networks before concluding in Chapter 12.

Chapter 2

Community Detection

In the Chapter 1, we introduced the idea of community detection. In this chapter, we formally define community detection, and present some of the theory belonging to the subject. We introduce the notion of graphs, and how they are used in community detection.

2.1 Definitions and notation

In the previous chapter, we wrote about finding the best grouping of a class of students. We wanted groups such that each student felt a connection to the group he or she is placed in. In computer science we often use graphs to model such data sets. Nodes are the entities of data, and the relationship between them are edges. In the class division problem, each student is an entity of data, and becomes a node. The friendship between students becomes edges. We define a graph $G = (V, E)$, V is the set of nodes and E is the set of edges. Further let n and m be the number of nodes and the number of edges respectively. The adjacency matrix of a graph G is denoted by A and the element A_{ij} is 1 if nodes i and j are neighbors, otherwise it is 0. The number of edges from node i is the degree of node i and is denoted k_i.

In community detection the goal is to divide data into groups. The graph we are working on should be divided into $C = \{c_1, c_2, , , c_r\}$ partitions, also called communities. The number of communities r is unknown when the algorithm starts. Each partition c_a is a set of nodes. In this thesis we will only be working with non-overlapping partitions. This means that one node can only be a member of one community such that $c_a \cap c_b = \emptyset$. Each node must be a member of some set, thus $\cup_{i=1}^{r} c_i = V$. The sum of the degrees of all nodes

inside a community is denoted k_c. The number of edges inside a community c is denoted by l_c.

2.2 Complex networks

In community detection we often consider complex networks. These networks represent systems or data, that is not random. It can origin from nature, society or almost anything. The classroom example from Chapter 1 is an example of a complex network. Here each node represents a person and there exist an edge between two nodes if the corresponding two persons are friends. Another famous example is depicted in Figure 2.1. The Zachary's karate club is a complex network that shows the relationships inside a karate club [9]. It consists of 34 nodes and 78 edges. The nodes represent the members of the karate club, and the edges represents which of the members interacted outside the club.

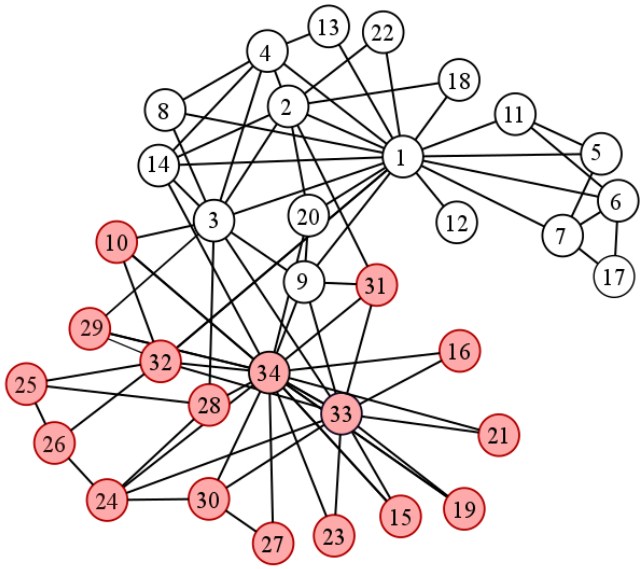


Figure 2.1: Zachary's karate club [9]

2.3 Finding community structure

It is often said that if there is a natural way to divide a graph into communities, then the graph contains "community structure". In many networks we know that the graph contains communities, and sometimes we are able to verify a

solution. This can be any natural affiliation between groups of nodes. If we look at all professional football players, we could have a natural community structure by placing all players in the same club in the same community. Or if one looks at a co-authorship graph, a natural community structure can be the different disciplines, or research fields. In these graphs with "ground-truth" communities, a solution can easily be verified or dismissed. The story behind Zachary's karate club depicted in Figure 2.1 is that Wayne W. Zachary studied a karate club over a period of three years. During the study, the president of the club and the instructor had a conflict. This led to the club splitting in two. Half of the members become students under the instructor, while the rest either found a new trainer or quit karate. In Figure 2.1, the instructor is node 1, and the president is node number 34. Here we can see a clear community structure. The members supporting the president forms one community, and the members supporting the instructor forms another community. The two communities are colored in the figure, where the red nodes supported the president, while the white nodes supported the instructor.

The challenging graphs are those without a known community structure. Here we do not have a simple way to verify if a solution is a good division of the graph. This problem exists because of the lack of a formal definition of what constitutes a community. In most natural networks it is infeasible to verify if a solution is correct. Imagine if we try to find the communities in a friendship graph. A natural community could be a group of friends where everyone knows everyone else (a clique). Asking each person of which group they should be placed in, would not give the answer. As with communities in general there is no formal definition on how densely connected a group of friends has to be.

2.4 Communities

So far we have only talked about communities as something vague like a group of data that has some similarities or some internal relationship. This is mainly because there does not exist any formal definition of a community. There has been a number of suggestions for how to define communities. So far none of these seems to have universally accepted. The early definitions were based on finding cliques, and clique like structures [37].

Another aspect of communities is whether they overlap or not. In a friendship graph, a person might feel that he or she belongs in several groups of friends. This is called soft clustering. A node representing a person could be included in several communities. Hard clustering is the case when we do not allow a node to be part of more than one community [14]. Thus soft and hard clustering represents non-overlapping and overlapping communities. As already stated in Section 2.1, in this thesis we only consider non-overlapping communities.

2.5 Community detection and graph partitioning

Both community detection and graph partitioning aims to split a graph G into smaller components. According to Newman [30], the main difference between these is that in a graph partition problem one almost always knows the number of partitions the graph should be split into, or the size of the partitions. Whereas in community detection this is unknown. An example is the (k, v) balanced partition problem. Here the goal is to split the graph into k components where size of each component is at most $v \cdot \frac{n}{k}$) [1]. In community detection we do not know the number of partitions, nor the size of any of the partitions.

2.6 Dense and sparse graphs

When working with networks we often characterize them as either dense or sparse. In a friendship graph it is possible that all people are friends, such that there are edges between all pairs of nodes. This is called a complete graph. A network that has close to the maximal number of edges is called a dense graph. A sparse graph is then a graph with significantly less edges than the maximum possible. There is no concrete definition on when a graph becomes sparse. This is often dependent on the context of the network. A graph is connected when there is a path between every pair of nodes. In the context of a friendship-graph this means that there exists a path from every person to any other person. In this thesis, we will only work with sparse connected graphs.

2.7 Summary

In this chapter, we introduced the mathematical notation used in community detection. We also introduced key aspects such as community structure, and complex networks. In the next chapter, we introduce some common features used in most community detection algorithms, and briefly explain some of these.

Chapter 3

Different approaches to community detection

In this chapter, we show some different approaches and algorithms to solve community detection. We start with introducing some common features many of today's community detection algorithms shares. Then we continue with briefly explaining some community detection algorithms, before finally justifying our decision to study the Louvain-method.

3.1 Common features in community detection algorithms

In this section we introduce some properties that are shared by many community detection algorithms. This includes both how they decide how a graph should be partitioned, and how the final result is returned.

3.1.1 Quality functions

When we partition a graph into communities, we need some way to say something about how a this division of the graph is. Not only do we need some way to determine if a partition is good, but we also need a way to decide which of two different partitionings that is the best. To do this we can use a "quality function". A function that maps a partitioning of a graph to a number. There exists several quality functions. For an overview of some of these see [7]. In Chapter 4, we take a closer look at modularity, one widely used quality function.

The fact that quality functions can make a prediction about which of two partitions that is the "best" is something that is utilized by several community detection algorithms. These algorithms usually try to maximize some numeric measure given by a quality function.

3.1.2 Hierarchical, agglomerative, and divisive algorithms

Some algorithms build a hierarchical structure of communities in the process of finding the best partitioning. This structure is often shown as a dendrogram. Hierarchical algorithms are often divided as being either agglomerative or divisive. Agglomerative algorithms can easiest be described as bottom-up algorithms, where the algorithm start with each node in a separate community. Throughout the algorithm communities are merged until there only exist one community containing all nodes. Divisive algorithms are top-down in the sense that they start with all nodes in the same community and then splits communities until each node are alone in separate a community. Some agglomerative and divisive algorithms return their "best" solution, and not a full dendrogram. This "best" solution is often selected using a quality function as described in Section 3.1.1. Figure 3.1 shows an example dendrogram with five nodes. The steps of the dendrogram are either merges or cuts depending on if an agglomerative or divisive algorithm is used.

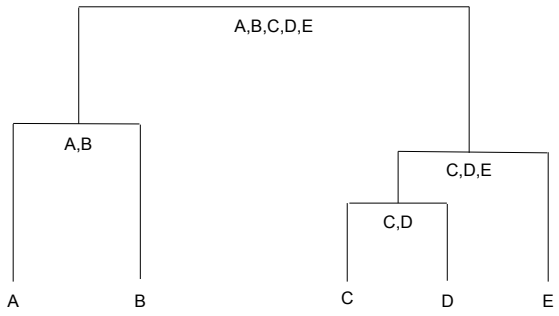


Figure 3.1: A dendrogram

3.2 Community detection methods

In this section we present some different community detection algorithms. These algorithms have been chosen since there exists implementations of these and they have all been tested thoroughly in an article by Yang et al. [39].

Edge Betweenness
Newman et al. have developed a community detection algorithm based on edge betweenness [17]. Freeman proposed that the betweenness centrality of a node i should be defined as the number of shortest paths between pairs of other nodes that run through i [16]. Newman et al. generalized this to also apply to edges. The edge betweenness of an edge is then the number of shortest paths between pairs of nodes that runs through it.

The algorithm calculates the edge betweenness for all edges, and then removes the edge with the highest betweenness. It then recalculates the betweenness for the edges, and remove the one with the highest betweenness. The algorithm continues to do this until all edges are removed. This results in a dendrogram as explained in Section 3.1.2. The algorithm runs time $O(m^2n)$ [17].

Greedy optimization of modularity
Clauset et al. developed an algorithm that tries to maximize the "modularity" a quality function [8]. The algorithm is a greedy hierarchical agglomeration algorithm. It starts off with each node in its own separate community. The algorithm then calculates the change in modularity for each pair of communities. The community pair with the highest change in modularity is merged together. Then we again calculate for all community pairs and again merge the two communities with the highest change in modularity. The algorithm continues as long as there is any pair of communities that give a positive change in modularity if they are merged. The algorithm returns a dendrogram, where the root is the best solution from the algorithm. The running-time is $O(md\log n)$, where d is the depth of the dendrogram[8].

Propagating labels
Raghavan et el. developed an algorithm using propagating labels [34]. This is very fast algorithm running in time $O(m)$ [34]. It starts with labeling each node with a unique label. It then iterates trough all nodes in a random order. Each node takes the same label as the majority of its neighbors with ties broken randomly. The algorithm stops when every node has the same label as the majority of its neighbors have. As the order in which the algorithm iterates trough the nodes and how ties are broken is random this method is not deterministic.

Leading eigenvector of the community matrix
This is another algorithm developed by Newman [29]. The algorithm uses the eigenvalues and eigenvectors of a modularity matrix. Using this matrix and its leading eigenvectors, the algorithm tries to maximize modularity. It stops once it is no longer able to increase the modularity. The running time is $O((m+n)n)$ [29].

Louvain-method
The "Louvain-method" was developed by Blondel et al. [5]. This method is a greedy agglomerative algorithm and is explained in detail in Chapter 5. The

running-time is estimated to $O(n\log n)$, but this has not been proven theoretically [38]. The algorithm also uses modularity as quality function.

Optimal modularity
Brandes et al. reduced modularity maximization to an integer linear programming problem, they then compute the optimal modularity value[6]. They also show that finding this value is NP-hard. Because of its exponential running-time, this algorithm is not able to handle large graphs.

Statistical mechanics
Reichardt et al. uses Potts model, and simulating annealing to partition a graph [35]. The running time is approximately $O(n^{3.2})$ for sparse graphs [10].

Random walks
Pons et al. used random walks to find communities [32]. The idea is that these random walks are likely to stay inside communities, as these should be densely connected. Starting from a totally non-clustered partition, the distances between all adjacent nodes are computed. Then, two adjacent communities are chosen and merged into a new community before the distances between the communities are updated. The running-time is $O(mn^2)$ [38].

3.3 Conclusion

In this chapter we have shown both some common features in community detection algorithms, and briefly explained some these algorithms. In Section 3.1.1, we introduced quality functions. Modularity is used as the quality function in many of the algorithms presented in this chapter.

We chose to describe these algorithms as they are all implemented and freely available. They have also been tested thoroughly in the article by Yang et al. [39]. In the article the authors compared the algorithms using benchmark graphs developed by Lancichinetti et al. [22]. They show how different algorithms perform on different types of graphs, and give recommendations on when to use the different algorithms. In almost all cases, they recommend using the Louvain-method. The Louvain-method uses modularity as quality function. In the next chapter, we explain modularity in detail. In the following chapters, we then introduce the Louvain-method, and how it maximizes modularity.

Chapter 4

Modularity

In this chapter we introduce and explain modularity, a quality function widely used in community detection. As we saw in Chapter 3, many known community detection algorithms uses modularity as their quality function. We start with the basis of modularity, and how it is calculated. We also outline some of the disadvantages of modularity. explains modularities disadvantages. Understanding modularity is crucial to understanding the Louvain-method, which is the subject of the rest of this thesis.

4.1 Introduction

Girvan and Newman introduced modularity as a quality function in 2004 [31]. As described in Section 3.1.1, a quality function maps a partitioning of a graph to a number. This lets us decide which of two partitionings is the best.

Modularity compares a given partition against a random graph. It measures how many edges there are inside each of our communities, compared to how many edges there would be inside the same community in a random graph. Graphs with high modularity are densely connected inside communities, and sparsely connected between communities. The number we get by calculating modularity is then simply a number measuring how dense the partitions are compared to a random graph.

4.2 The Modularity function

In a graph, the expected number of edges between two nodes i and j is $\frac{k_i k_j}{2m}$, where k_i and k_j are the degree of nodes i and j respectably. If there exist a node between two nodes, this is contained in the adjacency matrix element A_{ij}. In simple graphs A_{ij} is 0 if there does not exist an edge and 1 it it does. The Louvain-method uses weighted multi-graphs. This means that we allow the edges to have weight, and we allow self-loops. The element A_{ij} is then the weight of the edge between node i, and node j. If there does not exist an edge, the value is 0. The difference between the actual number of edges compared to the expected value is then:

$$A_{ij} - \frac{k_i k_j}{2m}.$$

Given an partitioning and summing over each node pair, we get the following formula for the modularity:

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{4.1}$$

Where c_i and c_j denotes the communities of nodes i and j respectably. The function δ is 1 if i and j are in the same community, otherwise it is 0. The values of Q is between -1 and 1.

The $\frac{1}{2m}$ part is merely conventional [30].

Modularity measures the difference between the fraction of edges in the graph that connects nodes inside the same community, against the expected value of the same node pairs in a graph with the same partitions, but with random connections between the nodes. A modularity of 0 indicates a completely random graph. We also achieve a modularity of 0 if all nodes are in the same community.

4.3 Modularity maximization

As modularity gives us a measure on how good our division of the graph is, we can try to choose a partition of the graph with the highest possible modularity. This is the idea behind modularity-maximizing community detection algorithms.

One obstacle to this is that finding the partition that gives the global maximum modularity is known to be NP-hard [6]. Because of this we cannot hope to find any polynomial time algorithms that is guaranteed to find the optimal

solution. Instead of looking for the global maximum, most community detection algorithms that uses modularity therefore tries to find some local maximum.

The Louvain-algorithm which we present in Chapter 5 utilizes the fact that it is easy and fast to calculate the change in modularity, when we move nodes between different communities.

4.4 The limitations of modularity

4.4.1 Resolution limit

It has been shown by Fortunato and Barthelemy that modularity suffers from a "resolution limit" [12]. This is simply the fact that community detection algorithms that utilizes modularity maximization as a goal, will sometimes merge communities that should be independent because this results in a higher modularity.

The impacts of the resolution limit can be easily shown by using a Caveman graph. A Caveman graph consist of n k-cliques. One edge is removed from each clique, and is instead used to connect the clique with the next one, such that all the cliques forms an unbroken loop. In figures 4.1 and 4.2, we have generated a caveman graph where $n = 30$, and $k = 5$. In Figure 4.1, each clique is a community, this results in a modularity of 0.867. If we merge two and two adjacent cliques into the same community, the modularity increases to 0.883. One could argue that these cliques should be in separate communities, as there is only one edge connecting them.

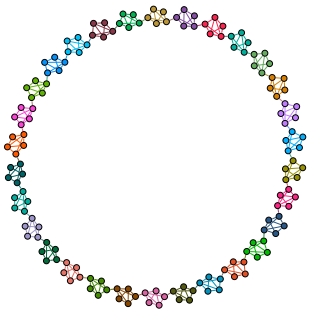


Figure 4.1: Q = 0.867

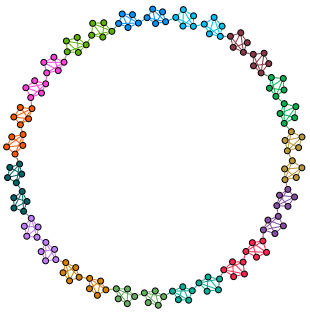


Figure 4.2: Q = 0.883

As a workaround some algorithms uses a resolution parameter to adjust for this. We can include the resolution parameter in the modularity function as: $Q = \frac{1}{2m}\sum_{ij}\left[A_{ij} - \gamma\frac{k_i k_j}{2m}\right]\delta(c_i, c_j)$ [35]. Here γ is the resolution parameter. Setting $\gamma < 1$ gives smaller communities, $\gamma = 1$ is the original definition of modularity, and $\gamma > 1$ gives larger communities. Blondel et al. showed that this comes at the expense of limiting the maximum size of the communities [21]. Thus it is possible to influence the size of the communities. The best value for μ is most likely dependent on the network we want to run our algorithm on. In this thesis, we will not use a resolution parameter.

4.4.2 Modularity in Random graphs

From the original modularity function, we should expect a modularity of 0 on random graphs. Guimera et al shows that there can exist partitions with high modularity in both random graphs and scale-free networks due to variations in the edge distribution [18]. They did this by generating random graphs, and then finding a partitioning of the graphs with high modularity.

4.5 Rewriting the modularity function

The original modularity equation 4.1 iterates all pairs of nodes. In this section, we show how we can rewrite this equation to iterate trough all communities instead. This lets us drop the δ function.

We can observe that there are only node-pairs within the same community that contributes to Q. Therefore we can rewrite the first part as:

$$\frac{1}{2m}\sum_{ij} A_{ij}\delta(c_i, c_j) = \sum_{c\in C}\frac{1}{2m}\sum_{i,j\in c} A_{i,j} = \sum_{c\in C}\frac{l_c}{m}$$

Where l_c is the number of links within community $c \in C$. The factor 2 disappears because each link is counted twice in A_{ij}.

The second part can be rewritten as follows:

$$\frac{1}{2m}\sum_{i,j}\frac{k_i k_j}{2m}\delta(c_i, c_j) = \sum_{c\in C}\frac{1}{(2m)^2}\sum_{i,j\in c} k_i k_j = \sum_{c\in C}\frac{k_c^2}{4m^2}$$

Here k_c is the sum of all degrees of the nodes in community $c \in C$.

By combining these two formulas, we get:

$$Q = \sum_{c \in C} \left[\frac{l_c}{m} - \left(\frac{k_c}{2m} \right)^2 \right] \tag{4.2}$$

4.6 Calculating modularity

In this section we show some examples of calculating modularity on simple graphs. In a triangle graph, there are three different possibilities for our partitioning. Either all nodes are in separate communities, or two are in the same community. The last option is all three nodes in the same community. All three different possibilities to partition a triangle graph are shown in figures 4.3, 4.4, and 4.5.

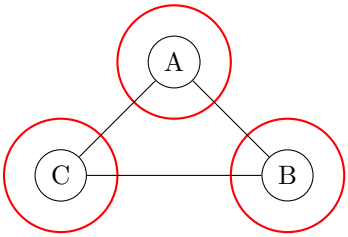


Figure 4.3: Three communities

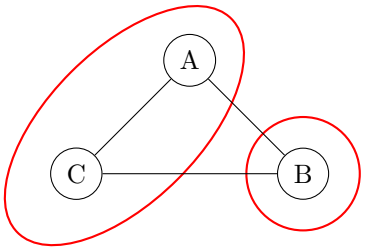


Figure 4.4: Two communities

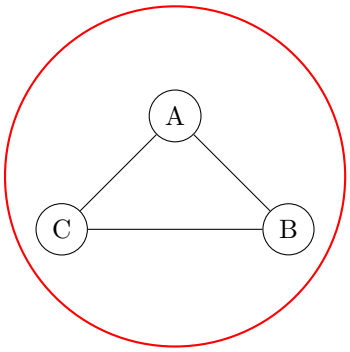


Figure 4.5: One community

Three communities With all three nodes in separate communities, we get:

$$Q = \sum_{c \in C} \left[\frac{L_c}{m} - \left(\frac{k_c}{2m} \right)^2 \right]$$

$$\begin{aligned} Q &= \left(\frac{0}{3} - \left(\frac{2}{6}\right)^2\right) + \left(\frac{0}{3} - \left(\frac{2}{6}\right)^2\right) + \left(\frac{0}{3} - \left(\frac{2}{6}\right)^2\right) \\ &= -\frac{1}{9} - \frac{1}{9} - \frac{1}{9} \\ &= -\frac{1}{3} \end{aligned} \tag{4.3}$$

Two communities In this alternative we calculate the modularity when we partition the graph with two of the nodes in the same community, and one node in the second community. It does not matter which two nodes we put in the same community as the graph is symmetric.

$$\begin{aligned} Q &= \left(\frac{1}{3} - \left(\frac{4}{6}\right)^2\right) + \left(\frac{0}{3} - \left(\frac{2}{6}\right)^2\right) \\ &= -\frac{1}{9} - \frac{1}{9} \\ &= -\frac{2}{9} \end{aligned} \tag{4.4}$$

One community With all nodes in one community, we get:

$$\begin{aligned} Q &= \left(\frac{3}{3} - \left(\frac{6}{6}\right)^2\right) \\ &= 1 - 1 \\ &= 0 \end{aligned} \tag{4.5}$$

As we can observe, in the triangle graph the maximum possible modularity we can achieve is 0.0. This happens when we have one community with all three nodes. The modularity will be 0 for any graph, when we have all nodes in the same community.

Newman showed that the modularity function (4.1) still holds for weighted graphs and multi-graphs [28].

4.7 Conclusion

In this chapter we explained the modularity quality function. Modularity gives us a numeric value for a partitioning. Using this number, we can compare two partitionings against each other. We also explained some of the disadvantages

of modularity such as the resolution limit and the possibility of getting high modularity in random graphs. In the next chapter, we introduce the Louvain-method and show how it uses modularity to find a partitioning.

Chapter 5

The Louvain-method

5.1 Introduction

In this chapter we introduce the Louvain-method, a greedy heuristic community detection algorithm. The method was first presented in the paper "Fast unfolding of communities in large networks" by Blondel, Guillaume, Lambiotte, and Lefebre [5]. All of the authors were connected to Université catholique de Louvain when the article was written. This is the basis for the nickname "the Louvain-method". We show how it uses the quality function modularity to compute a partition. We also show that the arxiv [4] version of the article contains an incorrect modularity equation.

5.2 Algorithm

The Louvain-method is a greedy agglomerative community detection algorithm that is based on modularity optimization. As mentioned in Chapter 4 maximizing modularity is NP-complete. The Louvain-method therefore finds a partition where the modularity is at a local maximum. The method consists of two phases that are repeated until the algorithm is unable to increase the modularity any further. We call one run of phase one and two, an iteration. For each iteration the algorithm builds a new level in a dendrogram, as seen in Figure 3.1. Each of these levels is a partitioning of the graph. The root of the dendrogram is the final partitioning, with the highest achieved modularity.

Phase one: The first phase is the modularity improvement phase. The phase starts with each node in a different separate community. The algorithm then

iterate trough all nodes in a random order. For each node i we calculate the potential change in modularity, when moving node i from its current community, to any of its neighbor's communities.

We then move node i from its current community, to the neighboring community that yields the highest increase in modularity. If none of the possible moves yield any increase in modularity, node i stays in its current community.

We continue iterating trough the nodes as long as there is any increase in modularity, or equivalently, as long as the algorithm performs at least one move per iteration trough the nodes. Phase one ends when there is no increase in modularity for an entire iteration trough the nodes. When we reach this state, we have found a partition giving a local maximum modularity.

Phase two: Phase two is the coarsening phase. Each community found in phase one is now contracted into a new node. Edges inside an old community is replaced with a self-loop attached to the new node. The weight of this self-loop is the sum of the weights of the replaced edges. All edges between two communities in the original graph are replaced with one new edge between the corresponding nodes. The edge weight of this new edge is equal to the sum of the weights of the replaced edges. These two phases are repeated until phase one is no longer able to obtain any more increase in modularity. Algorithm 1 shows the pseudocode of the Louvain-method.

Algorithm 1 Louvain-method

```
1: Let G be the initial network
2: while increase in modularity do
3:     Put each node of G in its own separate community
4:     while previous modularity < new modularity do
5:         for all nodes do
6:             Calculate move for node that yields highest increase in modularity
7:             if There exist a move a move with positive gain then
8:                 Move node to new community
9:             else
10:                Let the node stay in its current community
11:            end if
12:        end for
13:    end while
14:    if The new modularity is higher than the initial then
15:        contract G
16:    end if
17: end while
```

In figures 5.1, 5.2, and 5.3 we can see the different phases of the algorithm. Figure 5.1 shows a graph on which we want to run the algorithm. Figure 5.2 shows the result after the first phase. The different colors represents the communities found. In Figure 5.3 we have coarsened the graph. As we can see, we have replaced the three communities with three nodes, and replaced edges inside communities, and between communities with respectively self-loops, and edges between nodes.

Figure 5.1: Original graph

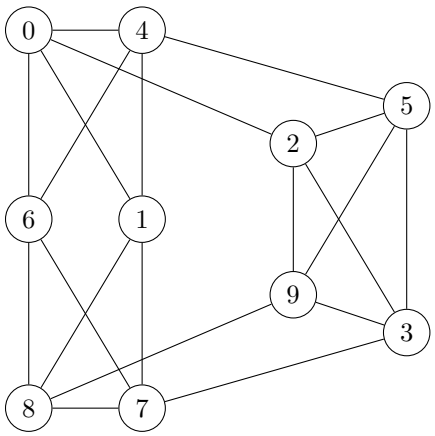


Figure 5.2: After first phase

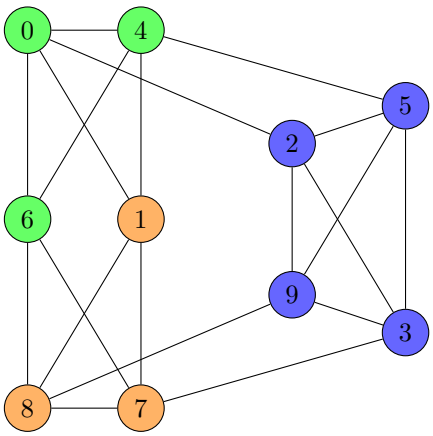
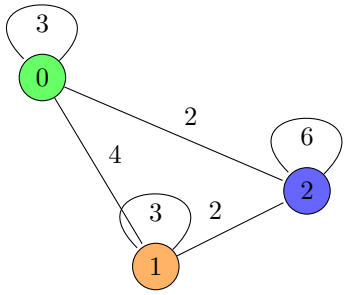


Figure 5.3: After phase two



5.3 Applications, and previous use of the algorithm

In the original article, the authors analyze a Belgian tele-network. They split the networks successfully into two main communities, one with French speaking customers, and one with Dutch speaking customers. The algorithm has also

been used to analyze among other Twitter [33], human brain functional networks [26], and Citation networks [40].

5.4 Modularity in the Louvain-algorithm

To calculate the modularity of a given partitioning of the graph we use the modularity equation (4.1) given in Chapter 4. To calculate the best move for a node, the Louvain-method uses the following equation to calculate the change in modularity, when moving an isolated node into a community [5]:

$$\Delta Q = \left[\frac{\sum_{in}+2k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \quad (5.1)$$

Let c be the neighbor community node i is merging into. Then $\sum_{in}$ is the sum of the weights of the edges inside c, $\sum_{tot}$ is the sum of the weights of the edges incident to nodes in c, k_i is the sum of the weights of the edges incident to node i, $k_{i,in}$ is the sum of the weights of the edges from i to nodes in c, and m is the sum of the weights of all the edges in the network.

5.4.1 Finding the best move

In the original paper describing the Louvain-method, the authors claimed that one in practice does not find the best move for each node by calculating the total change in modularity. Instead one should remove the node from its current community, and then evaluate the change in modularity of merging the node into all neighbor communities, including the previous community.

In this thesis, we use the total change in modularity to calculate the best move for a node. Equation (5.1) gives us the change in modularity, when merging an isolated node into a community. Let us denote the current node i and the current community of node i as a. We want to calculate the total change in modularity if we were to move node i into community b.

First, we need to calculate the change in modularity when we remove node i from community a, denoted $\Delta Q_{a,i}$. Then we calculate the change in modularity when merging node i into community b, denoted $\Delta Q_{i,b}$. We can use Equation (5.1) to calculate both $\Delta Q_{a,i}$ and $\Delta Q_{i,b}$. The value $\Delta Q_{a,i}$ is the same as negating the change in modularity of merging node i back into community a. This gives us an equation for the change in modularity when moving node i, from community

a to community b.

$$\Delta Q_{a,b} = \Delta Q_{i,b} - \Delta Q_{a,i} \tag{5.2}$$

5.5 Incorrect Δ Q - equation on arxiv

The original article presenting the Louvain-method "Fast unfolding of communities in large networks" is published on both arxiv [4], and in Journal of Statistical Mechanics: Theory and Experiment [5]. It turns out that the version published on arxiv has an incorrect Δ modularity equation. As the arxiv version is cited 5270 times, it is of interest to show why it is wrong. In [4] the following Δ modularity equation for merging an isolated node with another community, is used:

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

This can be simplified as follows:

$$\begin{aligned}
\Delta Q &= \left[\frac{\sum_{in}+k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \\
&= \frac{\sum_{in}+k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2 - \frac{\sum_{in}}{2m} + \left(\frac{\sum_{tot}}{2m}\right)^2 + \left(\frac{k_i}{2m}\right)^2 \\
&= \frac{\sum_{in}+k_{i,in}}{2m} - \left(\left(\frac{\sum_{tot}}{2m}\right)^2 + \left(\frac{k_i}{2m}\right)^2 + \frac{\sum_{tot} k_i}{4m^2} + \frac{k_i \sum_{tot}}{4m^2}\right) - \frac{\sum_{in}}{2m} + \left(\frac{\sum_{tot}}{2m}\right)^2 + \left(\frac{k_i}{2m}\right)^2 \\
&= \frac{\sum_{in}+k_{i,in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2 - \frac{\sum_{tot} k_i}{4m^2} - \frac{k_i \sum_{tot}}{4m^2} - \frac{\sum_{in}}{2m} + \left(\frac{\sum_{tot}}{2m}\right)^2 + \left(\frac{k_i}{2m}\right)^2 \\
&= \frac{\sum_{in}+k_{i,in}}{2m} - \frac{\sum_{tot} k_i}{4m^2} - \frac{k_i \sum_{tot}}{4m^2} - \frac{\sum_{in}}{2m} \\
&= \frac{\sum_{in}+k_{i,in}}{2m} - \frac{\sum_{in}}{2m} - \left(2\frac{\sum_{tot} k_i}{4m^2}\right) \\
&= \frac{\sum_{in}}{2m} + \frac{k_{i,in}}{2m} - \frac{\sum_{in}}{2m} - \left(\frac{2\sum_{tot} k_i}{4m^2}\right) \\
&= \frac{k_{i,in}}{2m} - \frac{\sum_{tot} k_i}{2m^2} \\
&= \frac{1}{2m}\left(k_{i,in} - \frac{\sum_{tot} k_i}{m}\right)
\end{aligned} \tag{5.3}$$

Using the original modularity equation (4.2), we can now show that Equation (5.3) is incorrect.

The change in modularity when merging two communities, a and b is given by:

$$\Delta Q_{ab} = \left[\frac{l_{ab}}{m} - \left(\frac{k_{ab}}{2m}\right)^2\right] - \left[\frac{l_a}{m} - \left(\frac{k_a}{2m}\right)^2\right] - \left[\frac{l_b}{2m} - \left(\frac{k_b}{2m}\right)^2\right]$$

Here l_a is the number of edges inside community a, $l_{ab} = l_a + l_b + l_{ab}$, and $k_{ab} = k_a + k_b$. l_{ab} is the number of edges between nodes in a and b.

In other words, we calculate how much the new merged community contributes to the sum, while subtracting how much the two previous communities contributed with.

In the Louvain-algorithm we are not merging entire communities, but instead merging one node into a community. The change in modularity when we merge community a, with node i will then be:

$$\Delta Q_{a,i} = \left[\frac{l_a + l_{i,a} + l_i}{m} - \left(\frac{k_a + k_i}{2m}\right)^2\right] - \left[\frac{l_a}{m} - \left(\frac{k_a}{2m}\right)^2\right] - \left[\frac{l_i}{2m} - \left(\frac{k_i}{2m}\right)^2\right]$$

Here we must add l_i because of possible self-loops.

$$\begin{aligned}
\Delta Q_{a,i} &= \frac{l_a + l_{i,a} + l_i}{m} - \left(\frac{k_a + k_i}{2m}\right)^2 - \frac{l_a}{m} + \left(\frac{k_a}{2m}\right)^2 - \frac{l_i}{2m} + \left(\frac{k_i}{2m}\right)^2 \\
&= \frac{l_a + l_{i,a} + l_i}{m} - \left(\frac{k_a^2 + 2k_a k_i + k_i^2}{4m^2}\right) - \frac{l_a}{m} + \left(\frac{k_a}{2m}\right)^2 - \frac{l_i}{2m} + \left(\frac{k_i}{2m}\right)^2 \\
&= \frac{l_a + l_{i,a} + l_i}{m} - \left(\frac{k_a^2}{4m^2} + \frac{2k_a k_i}{4m^2} + \frac{k_i^2}{4m^2}\right) - \frac{l_a}{m} + \left(\frac{k_a}{2m}\right)^2 - \frac{l_i}{2m} + \left(\frac{k_i}{2m}\right)^2 \\
&= \frac{l_a}{m} + \frac{l_{i,a}}{m} + \frac{l_i}{m} - \frac{k_a^2}{4m^2} - \frac{2k_a k_i}{4m^2} - \frac{k_i^2}{4m^2} - \frac{l_a}{m} + \left(\frac{k_a}{2m}\right)^2 - \frac{l_i}{2m} + \left(\frac{k_i}{2m}\right)^2 \\
&= \frac{l_a}{m} + \frac{l_{i,a}}{m} + \frac{l_i}{m} - \frac{k_a^2}{4m^2} - \frac{2k_a k_i}{4m^2} - \frac{k_i^2}{4m^2} - \frac{l_a}{m} + \frac{k_a^2}{4m^2} - \frac{l_i}{2m} + \frac{k_i^2}{4m^2} \\
&= \frac{l_{i,a}}{m} - \frac{k_a k_i}{2m^2} \\
&= \frac{1}{m}\left(l_{i,a} - \frac{k_a k_i}{2m}\right)
\end{aligned} \tag{5.4}$$

We can observe that Equation (5.4) is not equal to Equation (5.3). The difference shown in Equation 5.5. $\sum_{tot} = k_a$

$$\frac{1}{m}\left(l_{i,a} - \frac{k_a k_i}{2m}\right) \neq \frac{1}{2m}\left(k_{i,in} - \frac{\sum_{tot} k_i}{m}\right) \tag{5.5}$$

When we calculate the change in modularity for moving nodes, both equations will range possible moves for a node correctly. The most important consequence is that Equation (5.4) will return the exact change in modularity, while Equation (5.4) will not. In figures 5.4 and 5.5, we show an example of this. how the result from equation 5.3 is not correct.

Nodes with the same color, are in the same community. In the following we calculate the resulting change in modularity for merging node i into the green community.

Using Equation (5.3), we get:

$$\Delta Q = \frac{1}{20}\left(3 - \frac{9 * 5}{10}\right) = -0.075$$

When using Equation (5.4) we get:

$$\Delta Q = \frac{1}{10}\left(3 - \frac{9 * 5}{20}\right) = 0.075$$

By measuring the modularity before and after merging node i into the green community we can verify which of the equations that gives us the correct result.

$$\Delta Q = \text{modularity after} - \text{modularity before} = 0.265 - 0.19 = 0.075$$

As we can observe the equation 5.3 from [4] does not give the correct result.

Figure 5.4: Modularity = 0.19

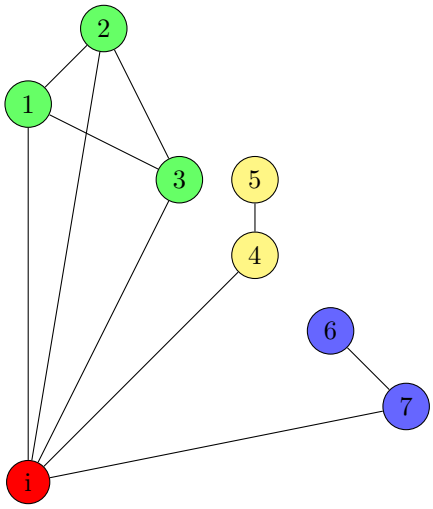
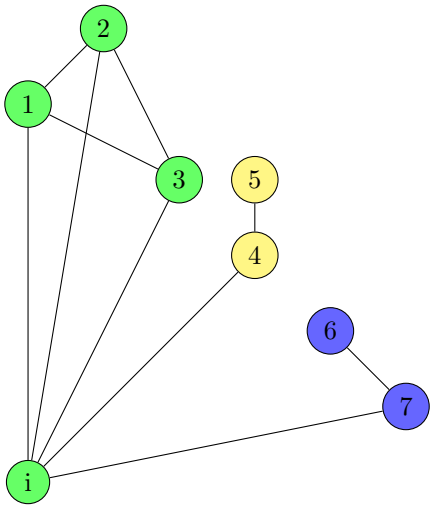


Figure 5.5: Modularity = 0.265



It turns out that the version of "Fast unfolding of communities in large networks" published on arxiv.org is not correct [4]. The authors of this article have also published a newer version [5], where they have corrected the Δ modularity equation.

5.6 Summary

In this chapter, we have introduced the Louvain-method. We have shown its structure, and how it maximizes modularity. In the next chapter, we take a closer look on how the algorithm works in practice. In particular, we will evaluate its running-time, number of iterations, and how the modularity increases throughout the execution of the algorithm.

Chapter 6

A closer look at the original algorithm

In this chapter we will take a closer look at how the Louvain-method perform in practice. We will observe how much of the work is done in the different iterations of the algorithm, and where the algorithm increases the resulting modularity the most.

6.1 Graphs

We have run the Louvain-method on several graphs. These graphs were collected from the Stanford SNAP library, the dimacs10 archive, and the UF Sparse Matrix Collection [25][11][36]. The motivation behind choosing these graphs has been the desire to have graphs in variating sizes, and containing community structure. Most of these graphs are based in real world networks. The selected graphs and some of their properties are listed in Table 6.1.

6.2 Hardware

All experiments have been performed on a computer with two Xeon E5-2699 v3 CPUs running at 2.30 GHz with 252GB ram. Each of the CPUs have 18 cores. As the implementation is strictly sequential, only one CPU has been utilized simultaneously.

Table 6.1: The different graphs

Name	N	E	<k>
karate	34	72	4.2
adjnoun	112	425	7.6
jazz	198	2742	27.7
celegans_metabolic	453	2025	8.9
email	1133	5451	9.6
power	4941	6594	2.7
PGPgiantcomo	10,680	24,316	4.6
as-22july06	22,963	48,436	4.2
Fe_rotor	99,617	662,431	13.3
preferentialAttachment	100,000	499,985	10
smallworld	100,000	499,998	10
com_dblp	317,080	1,049,866	6.6
com_amazon	334,863	925,872	5.5
com_youtube	1,134,890	2,987,624	5.3

6.3 My implementation

We have implemented the original Louvain-method and the variants described in Chapter 7. We have used Python 3 with the networkx library [19], a graph library for python. The implemetation is partly based on a reference implementation listed at the homepage of one the authors of the Louvain-method [3][2].

6.4 Results from the original algorithm

We have run the original algorithm on the graphs from 6.1. The results can be viewed in table 6.2. In this chapter, we take deeper look at the execution of the algorithm.

The number of iterations
We can observe from Table 6.2 that the algorithm does not use many iterations to terminate. The Jazz graph uses only one iteration, while the com-youtube graph uses seven. In the table, the graphs are sorted by the number of nodes. It looks like there is a trend that when we increase the number of nodes, the number of iterations also increase.

Table 6.2: Results from running the original algorithm

	Modularity	**Time(seconds)**	**#Iterations**
karate	0.429836397	0.005453825	2
adjnoun	0.210690655	0.058294535	2
jazz	0.247985962	0.171177626	1
celegans_metabolic	0.293300351	0.25210309	2
email	0.407092193	0.763338804	3
power	0.849030636	5.192291975	4
PGPgiantcompo	0.584299542	6.566255331	4
as-22july06	0.510704953	18.06820798	4
fe_rotor	0.905126941	510.9699843	4
preferentialAttachment	0.199042118	199.4465587	5
smallworld	0.745384929	77.62315607	5
com-amazon	0.92604453	209.2777104	5
com-dblp	0.821197814	315.6466172	5
com-youtube	0.712857656	1037.704317	7

6.4.1 Time usage

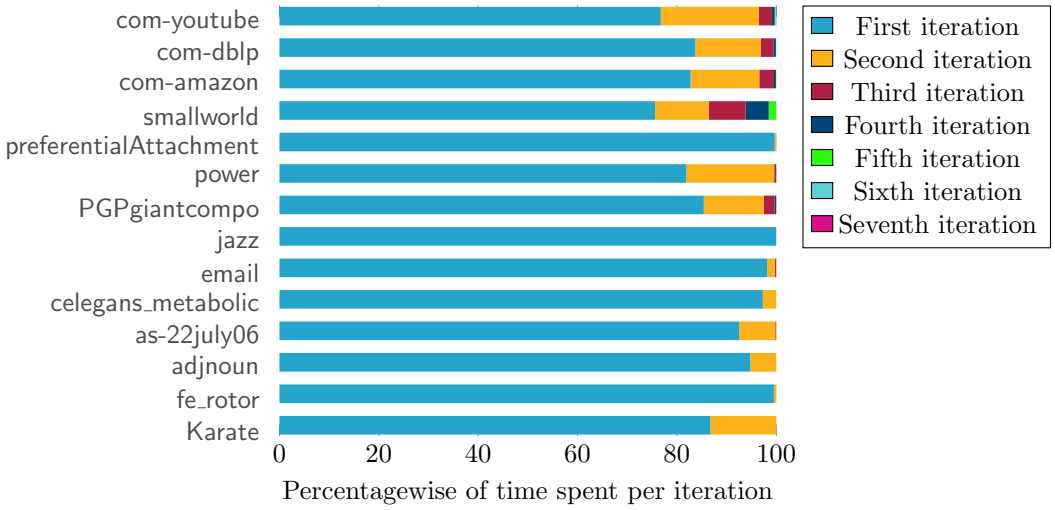
As expected, in Table 6.2 we can observe that the time spent on each graph increases as the number of nodes increase. There does exist some graphs that does not follow this trend. An example is easiest observed by looking at the preferential attachment, and small world graphs. Both graphs have $N = 100,000$, and $E \approx 500,000$. The small-world graph takes ~ 78 seconds, while the preferential attachment takes ~ 199 seconds. This suggests that there is something else than just the number of nodes and edges that decides the running-time of the Louvain-algorithm. This is most likely related to how distinct the community structure of a graph is. We can observe that the Louvain-method achieves a modularity ~ 0.75 on the small world graph, while only a modularity of ~ 0.2 on the preferential attachment graph.

We have also studied the running-time of each iteration of the algorithm. As we coarsen the graph for each iteration of the algorithm, there is clearly more work to do in the first iteration. The results can be viewed in Figure 6.1. On average over all graphs the first iteration uses 89.6% of the total running-time. The com-youtube graph has the highest number of iterations, and the last iteration uses only 0.07% of the total running-time.

6.4.2 Modularity increase

It is also interesting to see where the algorithm has the highest increase in modularity. In Figure 6.2, we can observe the change in modularity for each

Figure 6.1: Time usage in the Louvain-method per iteration



iteration. Like the running-time, most of the increase in modularity is achieved in the first iteration. The power graph stands out as an exception, where the increase in modularity is the highest in the second iteration. Overall for all graphs the percentagewise increase in modularity in the first iteration is 81.7%.

Modularity increase per node iteration

We have shown the increase in modularity per iteration of the Louvain-algorithm. As explained in Chapter 5, the Louvain-algorithm iterates trough all nodes, until an entire iteration trough the nodes does not result in any increase in modularity. We saw that the first iteration of the algorithm gave the highest increase in modularity. Here we show the increase in modularity for each iteration trough the nodes in the first iteration of the algorithm. The number of iterations trough the nodes the algorithm uses vary with every graph. This can be observed from Table 6.3. The Louvain-method uses only three iterations on the karate graph before finding a local maximum. The FE_rotor on the other hand uses 104 iterations. The result can be viewed in Figure 6.3. Here we have stacked the different iterations. Iteration one, two, three and four are independent, the rest are combined in groups. The average increase of the first iteration is 57.5% for the first iteration, 23.3% for the second, 8.9% for the third, 4% in the fourth, 5.4% for iterations 5 to 10, 0.4% for iterations 11-20, and for all following iterations.

Figure 6.2: Modularity increase per iteration

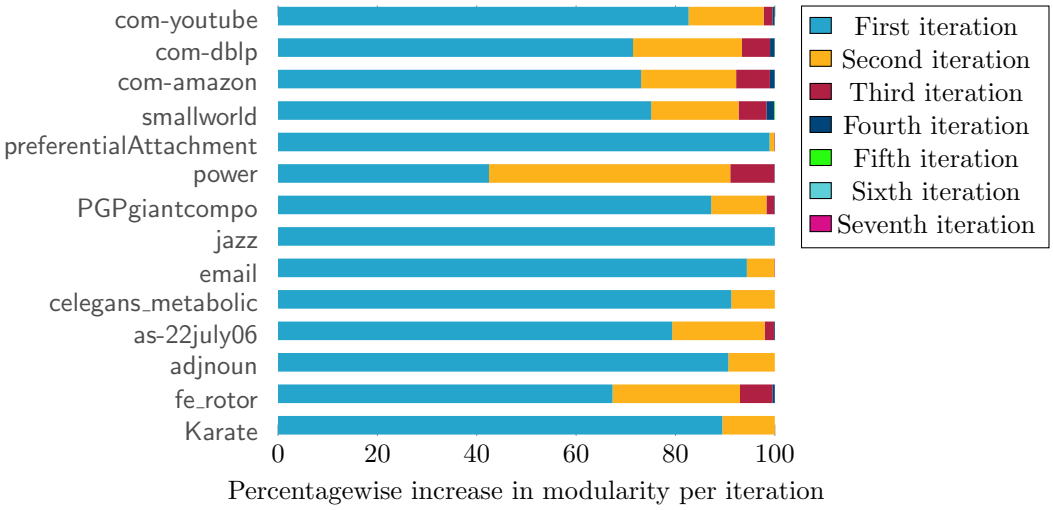
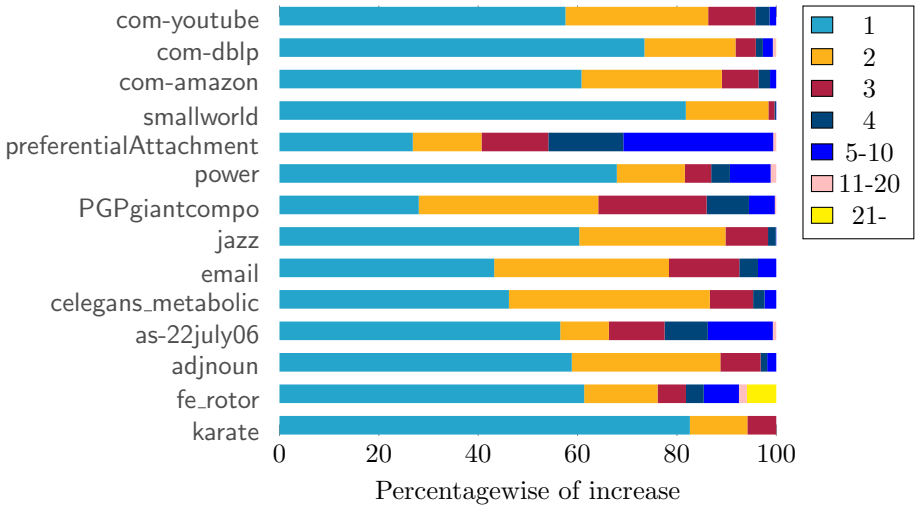


Table 6.3: The number of iterations trough the nodes

Graph	**Number of iterations trough the nodes**
karate	3
fe_rotor	104
adjnoun	6
as-22july06	16
celegans_metabolic	9
email	8
jazz	6
PGPgiantcompo	13
power	16
preferentialAttachment	27
smallworld	8
com-amazon	15
com-dblp	28
com-youtube	14

Figure 6.3: Modularity increase per iteration trough the nodes



6.5 Conclusion

In this chapter, we have shown the execution time of our implementation of the Louvain-method. We have also shown that the algorithm uses most of the running-time on the first iteration. The highest increase of modularity also happens in the first iteration. We have also shown that it is the first iteration trough the nodes in the first iteration that contributes with the highest increase in modularity. In the next chapter, we introduce several variants of the Louvain-method. The idea is that changing the order in which we traverse the graph the nodes could have an impact on running-time and the resulting modularity.

Chapter 7

Traversal orderings

7.1 Why different orderings?

The original algorithm iterates trough the nodes in a randomly chosen order. In this chapter will focus on the effect of using different orderings. The authors of the original article themselves raised the question whether the traversal order matter.

> "Preliminary results on several test cases seem to indicate that the ordering of the nodes does not have a significant influence on the modularity that is obtained. However the ordering can influence the computation time. The problem of choosing an order is thus worth studying since it could give good heuristics to enhance the computation time." [5]

In the following sections, we introduce several different orderings. For each traversal order, we have added a description and motivation behind the ordering. We have also included pseudo-code for each ordering. In Chapter 8, we run the orderings on graphs, and evaluate how they perform compared to the original Louvain-method.

7.2 Degree ranking 1

The degree ranking is quite straight forward. Instead of iterating trough the nodes in a random order, we sort the nodes based on their degrees. We have

chosen to sort them by decreasing degree. The idea behind this is that a node with a high degree is likely to be in the middle of some community. In some sense, it will be the hub of the community. Degree ranking can be implemented both with degree, and with weighted degree.

Algorithm 2 Degree ranking 1 - First Phase

```
1: Sort the nodes in a non-increasing order
2: while New Modularity > Previous modularity do
3:     for all nodes do
4:         Calculate best move, and move node
5:     end for
6: end while
```

7.3 Neighborhood degree ranking

The "neighborhood degree ranking" is similar to ordinary degree ranking, but instead of looking at just one node, we sum the degree of the node and all of its neighbors. This is implemented with both degree and weighted degree.

Algorithm 3 Neighborhood degree ranking - First Phase

```
1: Sort nodes by the combined degree of them self and their neighborhood
2: while New Modularity > Previous modularity do
3:     for all nodes do
4:         Calculate best move, and move node
5:     end for
6: end while
```

7.4 Neighborhood finder 1

In the "neighborhood finder" variant we start off with randomly selecting a node. We then calculate and make the best move for this node. We continue to do the same for all neighbors of the node. When all neighbors have been moved, we randomly select a new node and repeat the process. Each node we move is marked such that no node is moved twice in the same iteration. The idea behind this ranking is that nodes most likely belongs to the same community as their neighbors. This node ranking might find the "correct" communities faster, such that we don't move nodes back and forth between communities.

Algorithm 4 Neighborhood finder 1 - First Phase

```
1: while New Modularity > Previous modularity do
2:     for each node do
3:         if node is not marked then
4:             Mark node
5:             Calculate and perform best move for node
6:             for each neighbor of node do
7:                 if neighbor is not marked then
8:                     Mark neighbor
9:                     Calculate and perform best move for neighbor
10:                end if
11:            end for
12:        end if
13:    end for
14: end while
```

7.5 Neighborhood finder 2

This strategy is very similar to the previous "Neighborhood finder 1", but here we combine the degree ranking and the neighborhood finder. We start each iteration of the nodes by sorting them by descending, then for each node we move the node and its neighbor nodes. The idea is that the degree ranking gives us hub nodes, and the neighborhood finder merges them and their neighbors into a community. This is also implemented with both degree, and weighted degree. As with "Neighbourhood finder 1" each node is marked when moves, so that each node is only moved once per iteration through the nodes.

Algorithm 5 Neighborhood finder 2 - First Phase

```
1: while New Modularity > Previous modularity do
2:     Sort nodes in a non-increasing order
3:     for each node do
4:         if node is not marked then
5:             Mark node
6:             Calculate and perform best move for node
7:             for neighbors of node do
8:                 if neighbor is not marked then
9:                     Mark neighbor
10:                    Calculate and perform best move for neighbor
11:                end if
12:            end for
13:        end if
14:    end for
15: end while
```

7.6 Disturbed local maximum

When the original algorithm completes the first phase it has found some local maximal modularity configuration. The "Disturbed local maximum" randomly moves some of the nodes when this occurs. These random moves mean that we randomly choose to move each of the selected nodes to one of their neighbors communities, or stay in their community. After moving the nodes, we repeat phase one. The idea is that if we already had a good partition of the graph, the nodes will be moved back to their previous community. But if we did not have a good partition we might find a better now. We have chosen to move log N nodes.

Algorithm 6 Disturbed local maximum - First Phase

```
1: while New Modularity > Previous modularity do
2:     sort nodes in non-increasing order
3:     for each node do calculate best move, and move node
4:     end for
5: end while
6: Randomly select log N nodes, and perform a random move on them
7: while New Modularity > Previous modularity do
8:     Sort nodes by degree highest to lowest
9:     for each node do
10:        calculate best move, and move node
11:    end for
12: end while
```

7.7 Triangle ranking

The neighborhood degree ranking explained in Section 7.3 only considers the degree of a neighborhood. It does not say anything about how well connected the neighborhood is. In the triangle ranking we therefore calculate the number of triangles each node is a part of. In other words, the number of how many of the nodes neighbors that are also neighbors themselves.

Algorithm 7 Triangle ranking - First Phase

```
1: while New Modularity > Previous modularity do
2:     Sort nodes by the number of triangles they are part of
3:     for all nodes do Calculate best move, and move node
4:     end for
5: end while
```

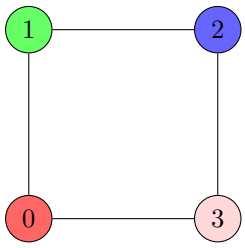
7.8 Modularity ranking 1

In the original algorithm, we iteratively go through the nodes and move the current node based on the highest possible increase in modularity. In the modularity ranking variant we calculate the best possible move for each node, without moving the nodes. We then sort the nodes based on their potential to increase the modularity in decreasing order. Then we move the node with the highest potential. For each move we make, we recalculate the list of best possible moves.

When moving a node, we do not need to recalculate the potential for all nodes. But we need to recalculate for all nodes in both the new, and the old community. We also need to recalculate for all neighbors of all nodes in these two communities.

In Section 5.4, we explained that the original article introducing the Louvain-method claimed that one in practice finds the best move for a node by removing the node from its current community, and then calculate the change in modularity for all neighbor communities, including the previous community. When we calculate and rank the nodes based on their potential to increase the modularity, this shortcut will not work.

Figure 7.1: Example graph



In the graph depicted in figure 7.1 all the nodes have the potential to increase the modularity equally much. If we were to use the suggested way of calculating the best move, this would result in an infinite loop. Let's say that we start with merging node 0 into community 1. In the next round, when we calculate the best move for each node, we would then again take node 0 out of its current community (1), and move it back.

If we set the restriction that moving back to the same community is not allowed, node 0 would just move between community 1 and 3. If we were to say that moving the same node twice is not allowed, then we would just switch between moving node 0, and 2 in and out of community 1. Therefore, we need to use the way described in section 5.4.1 to calculate the best move for the nodes.

Algorithm 8 Modularity ranking 1 - First Phase

```
1: Calculate best move for each node
2: Sort nodes by their potential to increase modularity
3: while There exist a node with a move that increases modularity do
4:     Move node to best community
5:     Recalculate and reorder all nodes by their potential to increase modularity
6: end while
```

7.9 Modularity ranking 2

As the first modularity ranking uses much time on calculating and recalculating best moves for all nodes it is not able to handle large networks. This variant calculates the best move for all nodes, and sort them once. We then iterate trough the nodes in this order. This makes the algorithm able to handle larger graphs, but still with a time penalty compared to the original algorithm.

Algorithm 9 Modularity ranking 2 - First Phase

```
1: while New Modularity > Previous modularity do
2:     Calculate best move for each node
3:     Sort nodes by their potential to increase modularity
4:     for all nodes do Calculate best move, and move node
5:     end for
6: end while
```

7.10 Conclusion

In this chapter, we have introduced several ways to traverse the nodes. The goal behind this these orderings is to reduce the number of "wrong" moves, such that a node does not switch between several communities, but is placed in the correct community at the first move. In Chapter 8, we have tested these traversal orderings, and compared them against the original algorithm.

Chapter 8

Results from the traversal orderings

In this chapter we present the results of testing the traversal orderings of the Louvain-method presented in Chapter 7. We show which variants that looks the most promising, both in regards of running-time, and on the resulting modularity. Throughout the chapter, we refer to the Louvain-method as described in the paper [5] as the "original algorithm".

8.1 Results

8.1.1 Modularity results

Figure 8.1 shows us the percentage difference in modularity the different orderings achieves compared to the original method. We can observe that most of the orderings does it both better and worse than the original algorithm depending on the graph. Table 8.1 shows the average percentagewise difference in modularity over all graphs compared to the original algorithm. We have implemented some of the variants with both degree, and wighted degree. In the variants where we use weighted degrees, we add "weighted" to the normal name.

Figure 8.1: The percentagewise difference in modularity compared to the original algorithm

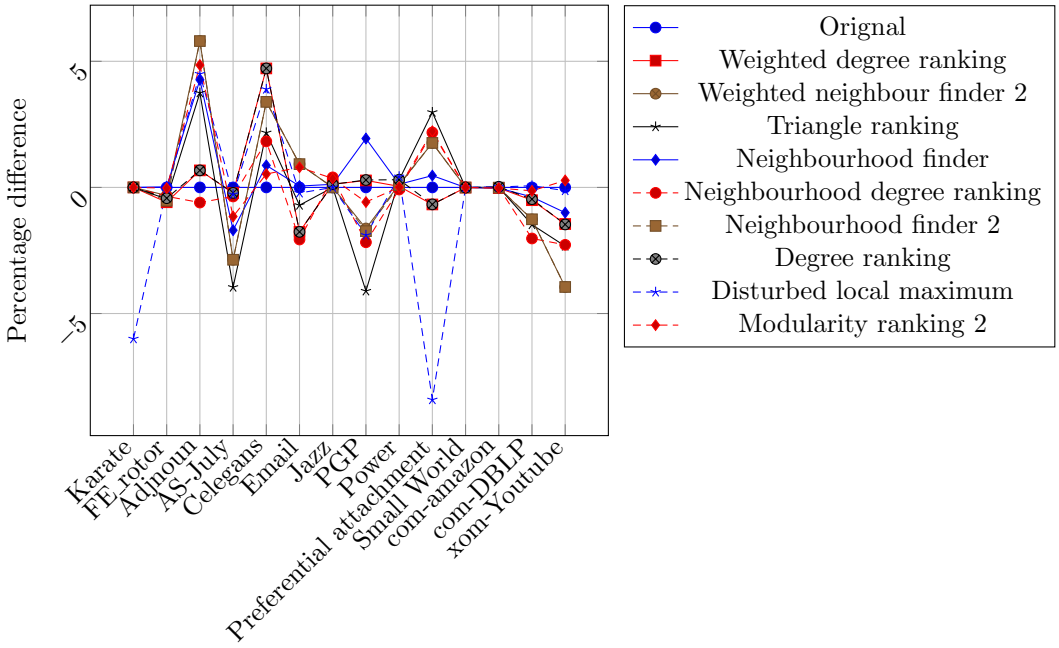


Table 8.1: Average percentagewise difference compared to the original algorithm

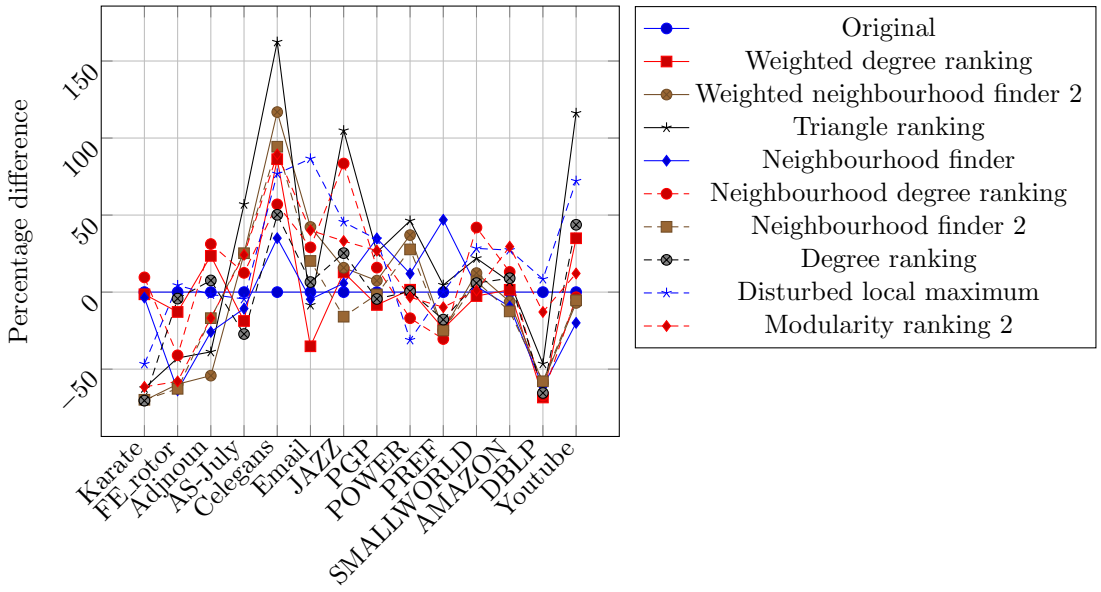
	Modularity	**Time**
Original	0	0
Weighted degree Ranking	0.04	-0.76
Weighted neighbor finder 2	0.14	-1.77
Triangle ranking	-0.28	24.78
Neighborhood finder	0.34	-4.37
Weighted neighborhood degree ranking	-7.39	86.25
Neighborhood degree ranking	-0.39	9.63
Neighborhood finder 2	0.12	-6.66
Degree ranking	0.08	-2.92
Disturbed local maximum	-0.58	77.52
Modularity ranking 2	0.51	6.72

8.1.2 Time results

Figure 8.2 shows us the percent difference in running time for each of the variants compared against the original algorithm. As we see most of the variants does

it both better and worse than the original algorithm depending on the graph. Table 8.1 shows us the average difference from the original algorithm.

Figure 8.2: Change in running-time compared to the original algorithm



8.2 Conclusion

Most of the orderings performs on average very similar to the original algorithm, in terms of modularity. From Table 8.1, we can observe that the worst variant has an average modularity $\sim 7\%$ lower than the original algorithm. The best variant achieve a slight increase in average modularity on $\sim 0.5\%$. This improvement is so small that we can not conclude that it is any significant change.

Considering the running-time there is some clearer differences. The "Weighted neighborhood degree ranking" is clearly the worst ordering with an average increase of $\sim 86\%$ in running-time compared to the original algorithm. Some of the other variants show some promise, as the "Neighborhood Finder 2" with an average running-time 6.7% lower than the original algorithm.

Out of the tested variations, we have chosen that "Modularity ranking 2", "Degree ranking", "Weighted degree ranking", "Neighborhood finder", "Neighborhood finder 2", and "Weighted neighborhood finder 2" looks the most promising. This is based on two factors. They either achieve a higher modularity, or they

have a lower running-time than the original algorithm.

In the Chapter 9, we introduce a modularity threshold, and different variations of this threshold. We test how these thresholds affect the running-time and achieved modularity with the original algorithm. In Chapter 10, we test the selected orderings, with these thresholds and observe how they perform compared to the original algorithm.

Chapter 9

Threshold variations and results

In Chapter 6 we saw that most of the time was spent on the first iteration of the algorithm. Most of the increase in modularity was also achieved in this iteration. In this chapter, we introduce modularity thresholds, and look at how variations of this effects the running-time and the resulting modularity.

9.1 Experimenting with different thresholds

The original algorithm iterates trough the nodes until it is not possible to find any move that results in a positive change in modularity. We have then found a local maximum in modularity, without any possible positive moves. As we saw in Chapter 6, most of the modularity increase comes in the first iteration of the original algorithm, and also in the first iterations trough the nodes.

The original implementation of the Louvain-method, used in the original article introducing the Louvain-method uses a modularity threshold [24].

> "It is interesting to note that the speed of our algorithm can still be substantially improved by using some simple heuristics, for instance by stopping the first phase of our algorithm when the gain of modularity is below a given threshold...The impact of these heuristics on the final partition of the network should be studied further, as well as the role played by the ordering of the nodes during the first phase of the algorithm." [5]

The threshold they used in the implementation works by checking how much the modularity have changed each time we iterate trough the nodes in phase one. If the change is bellow some given threshold, the algorithm end the first phase and continue with phase two. In the article, they did not specify which value they used for the threshold.

9.2 Single threshold

In this section we take a closer look at the running time and the resulting modularity, when we add a single threshold to the original algorithm. This threshold variant is equal to the one they have implemented in the original implementation of the Louvain-method. The threshold is implemented by checking how much the modularity have changed each time we have iterated trough the nodes in phase one. If the change is below the given threshold, the algorithm end the first phase and continue with phase two. The threshold remains the same through all iterations of the algorithm.

We have run the original algorithm with several thresholds. The results for each graph are plotted in figures 9.1 and 9.2. All the presented results are the percentagewise difference from the original algorithm without a threshold. The average difference over all graphs is shown in Table 9.1.

Table 9.1: Average percentagewise change in modularity and running-time single threshold

Threshold	**Modularity**	**Running-time**
0.0001	0.26	-23.77
0.001	0.03	-40.34
0.01	0.02	-57.65
0.05	-11.08	-74.01
0.1	-14.58	-77.21
0.2	-36.71	-80.47
0.3	-46.24	-79.64
0.4	-52.16	-79.48
0.5	-53.42	-84.17
0.6	-56.25	-84.45

Figure 9.1: Percentagewise change in modularity compared to the original algorithm

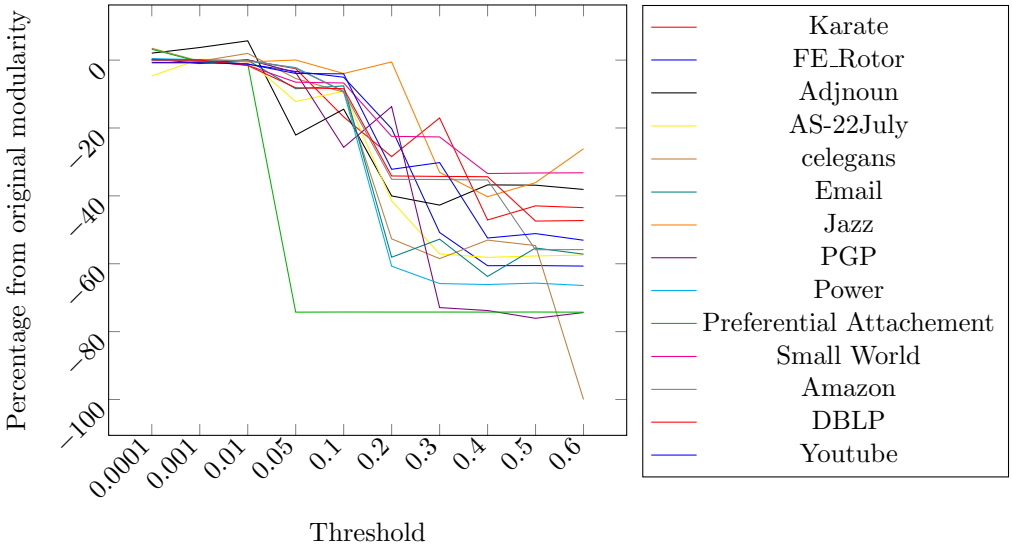
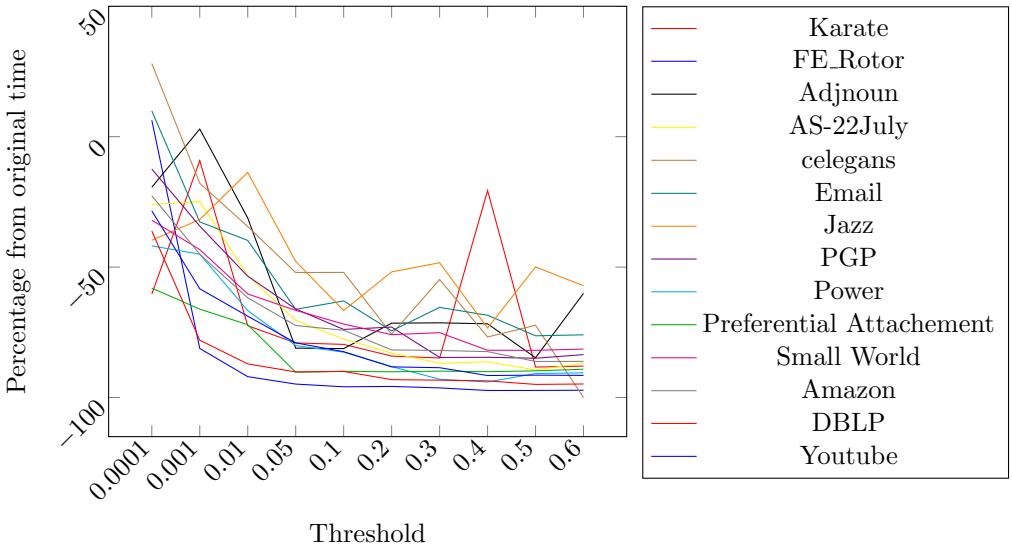


Figure 9.2: Percentagewise change in running-time compared to the original algorithm



We can observe that both the resulting modularity and the running-time drops when we increase the threshold value. From Table 9.1, we can observe that

a threshold above 0.01 gives a drastic reduction in the achieved modularity compared against the original algorithm without a threshold. With a threshold of value 0.05, we get an average reduction in modularity of -11%. The highest threshold we tested was 0.6, here the reduction in average modularity was -56% compared to the original algorithm. Thresholds in the range 0.0001 to 0.01 achieved very similar results as the original algorithm.

The reduction in running-time is significant for all the thresholds we tested. Even with a threshold of 0.0001, we achieve an average reduction in running-time of -23.8% compared to the original algorithm. Surprisingly we can observe a small increase in modularity for low thresholds.

9.3 First iteration threshold

In this section we build upon the results from Section 9.2. We saw that by adding a modularity threshold, we could reduce the running-time, but the achieved modularity was also decreased. In the "first iteration threshold" variant we apply a threshold for just the first iteration of the algorithm. The idea is that since the first iteration uses the most time, this is where we should focus on reducing the running-time. In the following iterations, the graph has been coarsened into a much smaller graph and we do therefore not need to reduce the running-time of these iterations.

The results can be viewed in figures 9.3 and 9.4. For all experiments, we have varied the threshold from 0.0001 and up to 0.6. The following iterations is as in the original algorithm without any threshold. As in the previous section the plots show the percentage difference from running the modified algorithm compared to the original algorithm. Table 9.2 shows the average percentagewise difference between the original algorithm and the modified algorithm. We can observe that for thresholds with values in the range $0.05 \rightarrow 0.6$ the average difference compared to the original algorithm is much lower than when using a single threshold on all iterations.

Table 9.2: Average change in modularity and running-time first iteration threshold compared to the original algorithm

Threshold	Modularity	Time
0.0001	-0.11	-19.82
0.001	0.67	-43.32
0.01	-0.09	-51.33
0.05	-1.03	-58.66
0.1	-1.15	-61.62
0.2	-2.49	-58.51
0.3	-2.49	-54.56
0.4	-2.86	-57.29
0.5	-3.58	-53.36
0.6	-2.65	-58.29

Figure 9.3: Percentagewise change in modularity compared to the original algorithm

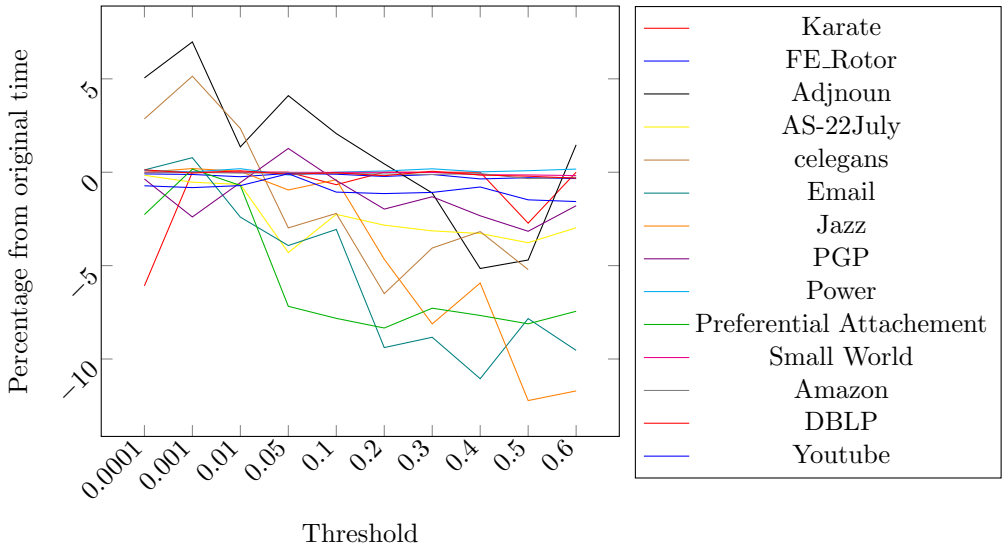
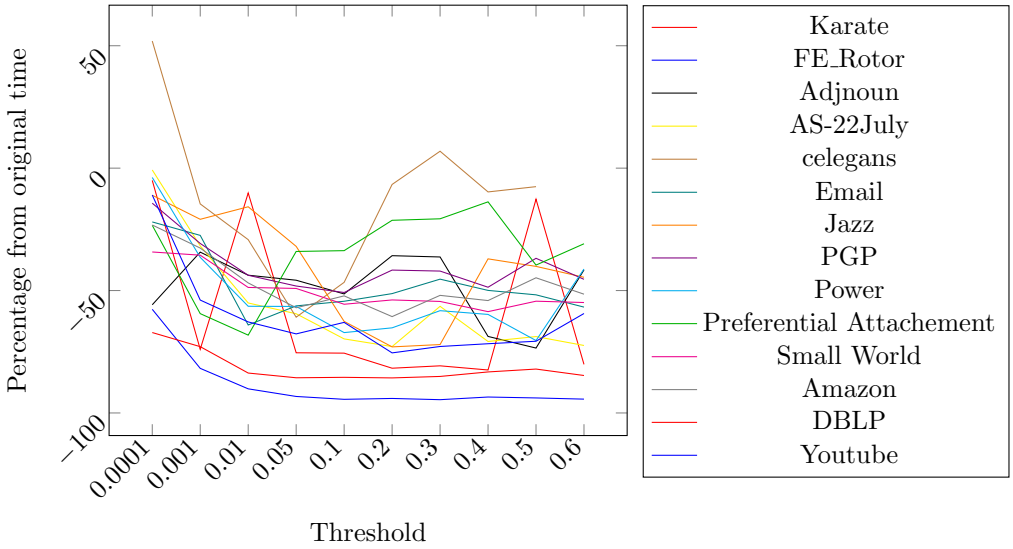


Figure 9.4: Percentagewise change in running-time compared to the original algorithm



9.4 Divided threshold

In the "divided threshold" variant we have a separate threshold for each iteration. The way this is done is by giving the algorithm one more parameter. In addition to a threshold, we also add a dividend. For each iteration, the current threshold is divided by the dividend. This result in a descending threshold for each iteration. In figures 9.5, and 9.6, we have plotted the average percentage difference compared to the original algorithm for each threshold over all the tested graphs. We can observe that the running-time quickly decreases, and the modularity stays high at low thresholds. Not surprisingly we can also observe that a high threshold, and low dividend gives a large reduction in modularity.

Figure 9.5: Average percentagewise change in modularity compared to the original algorithm

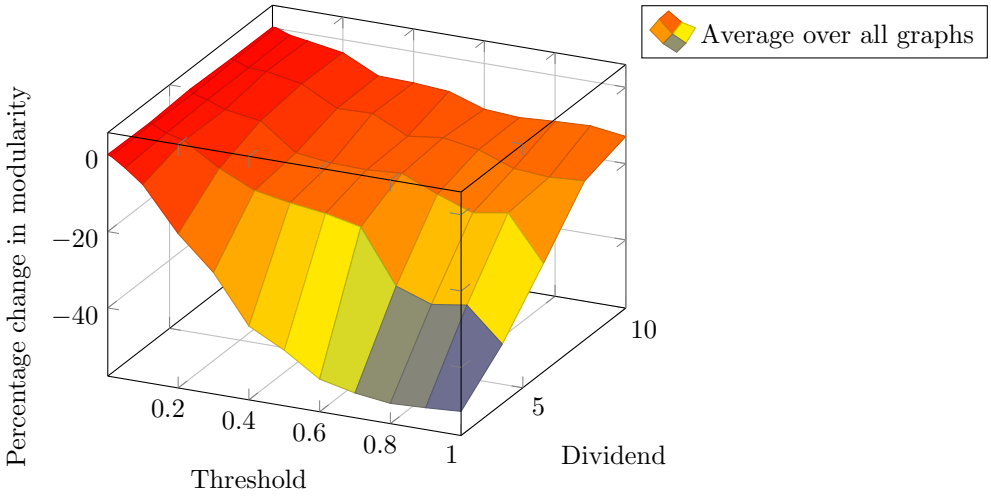
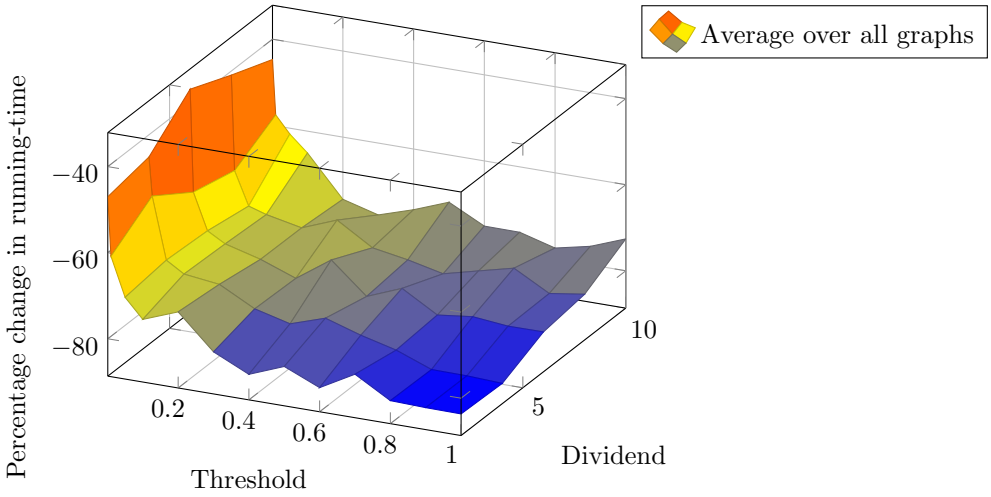


Figure 9.6: Average percentagewise change in running-time compared to the original algorithm



9.5 Random variant

The last variant in this chapter is having a random variant. Here we have a random phase on in the first iteration. This is done by iterating trough the nodes,

and randomly merging each node with one of its neighboring communities, or letting it stay in its current community. This is done to see how much the first iteration affects the resulting modularity, or if the following iterations could be able to counteract the random phase.

The results are given in Figure 9.3. Each value is the percent difference from the original algorithm. The running-time is on average 61.9% lower than the original algorithm. While this is a major speed-up, the 20.8% reduction in modularity is not tolerable. The modularity reduction shows that the random first iteration variant should not be pursued further.

Table 9.3: Results from random first iteration

	Modularity	**Time**
Karate	-39.11	-86.36
fe_rotor	-1.48	-87.49
adjnoun	-29.91	-73.08
as-22july06	-15.01	-81.59
celegans_metabolic	-29.47	-27.58
email	-33.66	-52.68
jazz	-58.91	-34.03
PGPgiantcompo	-15.56	-49.42
power	-1.08	-76.12
preferentialAttachment	-25.67	-56.87
smallworld	-26.29	-20.59
com-amazon	-2.29	-58.63
com-dblp	-5.24	-81.04
com-youtube	-7.72	-81.39
Average:	**-20.81**	**-61.92**

9.6 Summary

In this chapter, we have shown different types of thresholds, and shown how these affect both the running-time and the achieved modularity. The presented results show that introducing a threshold causes a drastic reduction in running-time. The first iteration threshold, and the divided threshold performed much better than the single threshold variant on high threshold values. On lower values, they all performed very similar. In the first iteration threshold, it looks like a threshold value between 0.001 and 0.1 is the most optimal regarding modularity and running-time. In this range, we achieve a reduction in running-time of 40-60% with a reduction in modularity of only 0-1.1%. In the divided threshold, it looks like the same range of thresholds with a dividend higher than 8 achieves nearly the same values.

In the next chapter, we test how some of the traversal orderings from Chapter 7 perform when we modify them with thresholds.

Chapter 10

Combining sorting variants with modularity threshold

In this chapter, we combine the traversal orderings introduced in Chapter 7, with the thresholds introduced in Chapter 9. Most of the different orderings did not show any significant increase in modularity compared to the original algorithm, and some had a higher running-time. In this chapter, we therefore examine if there is anything to achieve by adding modularity thresholds to the orderings. This will most likely reduce the running-time. The question is then whether the resulting modularity stays at the same level as the original algorithm, or higher.

10.1 Combined variants

From Chapter 7 we have chosen to continue testing Modularity ranking 2(MR2), Degree ranking(DR), Weighted degree ranking(WDR), Neighborhood finder(NF), Neighborhood finder(NF2), and Weighted Neighborhood Finder 2(WNF2). These orderings looked the most promising. In the Chapter 9, we introduced single, first iteration, divided, and random threshold types. We have chosen to combine the mentioned orderings with both first iteration and divided thresholds. With first iteration threshold, we have used threshold values 0.001,0.01, 0.05, and 0.1. With divided threshold, we use threshold values 0.01, 0.05, and 0.1, and dividends 6, 8, and 10. With 13 threshold variations, and 6 orderings, we test a total of 78 different variants.

10.2 Results

In tables 10.1, and 10.2, we can see the result of running the selected variants with the selected thresholds. All entries are the percentagewise difference from the original algorithm. We show the average over all tested graphs. The list of graphs can be viewed in Table 6.1. The different types of thresholds are denoted *threshold/divider* for the divided thresholds, and **FI**threshold for the first iteration thresholds.

Table 10.1: Percentagewise modularity compared to the original algorithm

Threshold	MR2	DR	WDR	NF	NF2	WNF2
0.01/6	-0.84	-0.17	-0.17	0.22	-0.32	-0.31
0.05/6	-1.58	-2.44	-2.45	-1.55	-1.29	-1.29
0.1/6	-1.88	-4.05	-4.02	-2.24	-2.47	-2.62
0.01/8	-0.80	-0.16	-0.17	-0.52	-0.28	-0.26
0.05/8	-1.36	-2.38	-2.40	-2.03	-1.25	-1.25
0.1/8	-1.85	-2.58	-2.61	-2.65	-2.28	-2.43
0.01/10	-0.80	-0.15	-0.17	-0.85	-0.27	-0.26
0.05/10	-1.36	-2.22	-2.23	-1.69	-1.22	-1.22
0.1/10	-1.84	-2.55	-2.61	-2.94	-2.28	-2.39
FI0.001	0.46	0.08	0.04	0.63	0.02	0.02
FI0.01	-0.67	-0.13	-0.16	-0.28	-0.27	-0.26
FI0.05	-1.13	-2.05	-2.01	-1.33	-1.08	-1.07
FI0.1	-1.36	-2.27	-2.31	-2.45	-1.68	-1.78

Table 10.2: Percentagewise running-time compared to the original algorithm

Threshold	MR2	DR	WDR	NF	DNF	WDNF
0.01/6	-71.35	-87.01	-75.23	-79.24	-79.51	-76.71
0.05/6	-75.25	-90.00	-80.60	-81.60	-83.87	-80.48
0.1/6	-77.46	-91.03	-82.55	-85.44	-85.83	-83.07
0.01/8	-71.17	-86.96	-75.66	-77.94	-79.34	-76.56
0.05/8	-75.93	-89.43	-80.14	-83.47	-82.71	-80.41
0.1/8	-77.34	-90.34	-81.62	-85.49	-84.76	-82.07
0.01/10	-69.84	-86.77	-74.32	-79.69	-78.87	-75.53
0.05/10	-74.40	-89.56	-80.36	-83.46	-82.34	-79.61
0.1/10	-76.45	-90.67	-81.70	-84.04	-84.01	-81.70
FI0.001	-60.19	-81.45	-81.79	-64.98	-79.46	-68.88
FI0.01	-70.38	-85.98	-85.49	-74.68	-84.04	-75.96
FI0.05	-73.46	-88.21	-86.84	-77.15	-85.40	-77.76
FI0.1	-74.31	-88.03	-85.27	-78.86	-86.13	-78.79

Table 10.1 shows us that all the chosen orderings perform quite similarly as the original algorithm in regards of modularity. Most table entries are between 0 to -2%. The worst result is degree ranking with threshold 0.1/6, this variant has a reduction in average modularity of -4.05% compared to the original algorithm. Table 10.2 shows a drastic reduction in running-time compared to the original algorithm. The lowest reduction achieved was 60.19%, while the highest was 90.67%.

In Chapter 9, we used thresholds on the original algorithm. Table 10.3 shows the results from adding the same thresholds from this chapter to the original algorithm. If we compare the results from Table 10.3, and the results from tables 10.1 and 10.2, we see that the orderings from Chapter 7 performs very similarly with regards to modularity, and have a little lower running-time.

Table 10.3: Percentagewise difference when adding thresholds to the original algorithm

Threshold	**Modularity**	**Time**
0.01/6	-0.39	-60.61
0.05/6	-1.39	-69.05
0.1/6	-3.21	-71.39
0.01/8	0.09	-63.01
0.05/8	-1.39	-70.71
0.1/8	-1.54	-71.99
0.01/10	0.18	-57.45
0.05/10	-1.14	-61.36
0.1/10	-1.88	-65.08
FI0.001	0.67	-43.32
FI0.01	-0.09	-51.33
FI0.05	-1.03	-58.66
FI0.1	-1.15	-61.62

10.3 Summary

In this chapter, we have shown how the traversal orderings combined with thresholds performs compared to the original algorithm. Overall, they achieve a similar modularity with percentagewise results in the range 0.63 to -2.94%. Their running-time is significantly lower than the original algorithm, with reductions in the range $\sim 60 - 90\%$ compared to the original algorithm. In the next chapter, we test some of these variants on community detection benchmark graphs.

Chapter 11

Benchmark graphs

11.1 Intro

In complex networks the true community structure is often not known. This makes it difficult to say how good a partitioning of a graph is. In Chapter 4, we introduced modularity, a widely used quality function. Having a quality function helps us find the community structure, but may have disadvantages. An example is the "Resolution limit" of modularity presented in Section 4.4. This disadvantage show that modularity has a bias in the way it finds communities.

Benchmark graphs gives us the possibility to test our algorithms on graphs, where the community structure is known. This makes it possible to compare different algorithms with different quality functions against each other. In this chapter we will explain Girvan-Newman benchmark graphs (GN) [17] and Lancichinetti–Fortunato–Radicchi benchmark graphs (LFR) [23]. We will also run our final variants on some LFR-graphs, and evaluate the results.

11.2 Girvan-Newman Benchmark (GN)

Girvan and Newman introduced a type of benchmark graphs to evaluate the performance of their algorithm [17]. It has later become known as Girvan-Newman benchmark graphs. These graphs consist of 128 nodes, divided over four communities with 32 nodes in each community. Different graphs are created by changing how densely connected the four communities are. We create a GN-benchmark graph by creating all 128 nodes, and dividing them into the four communities with 32 nodes in each community, then we add the edges. For

each node pair inside the same community the probability for adding an edge is P_{in}, node pairs in different communities are connected with a probability P_{out}. P_{in} and P_{out} should be chosen so $P_{in} > P_{out}$. The probabilities should be chosen such that the average degree is 16. Figure 11.1 shows an example of an GN benchmark graph.

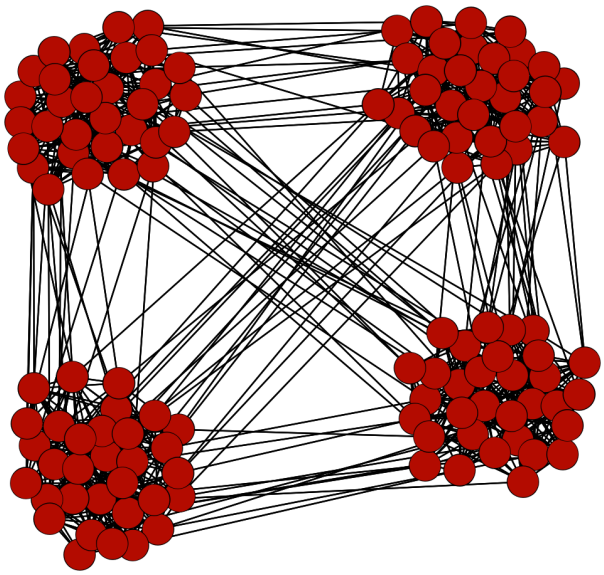


Figure 11.1: Girvan-Newman benchmark graph

11.3 Lancichinetti, Fortunato, and Radicchi benchmark

The GN-benchmark graphs give us a known community structure, but has some limitations. There are only four communities, each of the same size. In real world networks, equal sizes of communities are not common, in addition 128 nodes is much smaller than most networks. In 2008 Lancichinetti, Fortunato, and Radicchi(LFR) introduced a new type of benchmark graphs [23]. In these graphs the degree distribution and the community sizes follows power laws, respectively with exponents γ_1 and γ_2. Having a degree distribution following power laws, means the probability that a node has the degree k is given by $p(k) = k^{\gamma_1}$. Each node shares a fraction $(1 - \mu)$ of its edges with nodes in the same community, and a fraction μ with nodes in other communities. μ is called the mixing parameter.

To generate an LFR graph, we can use the following steps [13]:

1. We create a sequence of community sizes. This is done by randomly choosing numbers from a power law distribution with exponent γ_2.
2. Each node is given a internal degree of $(1-\mu)k_i$. The degree of each node is decided by the degree distribution following $k^{-\gamma_1}$
3. Nodes in the same community are randomly connected, every node are connected with a number of nodes equal to its internal degree.
4. Each node is given an external degree equal to μk_i. Each node is randomly connected with nodes in other communities until each node i are connected with k_i other nodes.

The LFR benchmark graphs gives us graphs with variable community sized and node degrees. This is more similar to real world networks than the G-N benchmark graphs.

Normal mutual information
When we test out Louvain-method variations on LFR-graphs, we need to be able to measure the result. To do this we can use normal mutual information(NMI). When we create an LFR graph, we also get the correct partitioning of the graph. We can use NMI to compare this partitioning with the partitioning we get from running the Louvain-method. We have used the same function to calculate the NMI as Yang et al. in [39].

11.4 Results

In this section, we will test some of the methods from Chapter 10 on some LFR graphs with variable μ. In Chapter 3, we mentioned an article by Yang et al., here they test several community detection algorithms using LFR graphs [39]. We have chosen to use the same parameters as they to generate LFR-graphs, given in Table 11.1. To create the LFR graphs, we have used a graph generator created by Fortunado et al. available at [15].

We have chosen to test several of the traversal orderings from Chapter 7. We have tested the original algorithm, both with and without thresholds, Degree ranking, Neighborhood finder, and Neighborhood finder 2. We use thresholds $FI0.001$, $FI0.01$, 0.01/10 We have chosen these variants and thresholds as they performed well compared to the original algorithm in Chapter 10. The results are given in figures 11.2, 11.3, and 11.4. In these plots, we have not plotted the relative value compared to the original algorithm, but instead plotted their actual values.

Table 11.1: Parameters used to generate the LFR-graphs

Parameter	**Value**
Number of nodes N	30 000
Maximum degree	0.1N
Maximum community size	0.1N
Average degree	20
Degree distribution exponent	-2
Community size distribution exponent	-1
Mixing parameter μ	0.03, 0.1,0.2,0.3,0.4,0.5,0.6,0.7

Figure 11.2: Achieved modularity

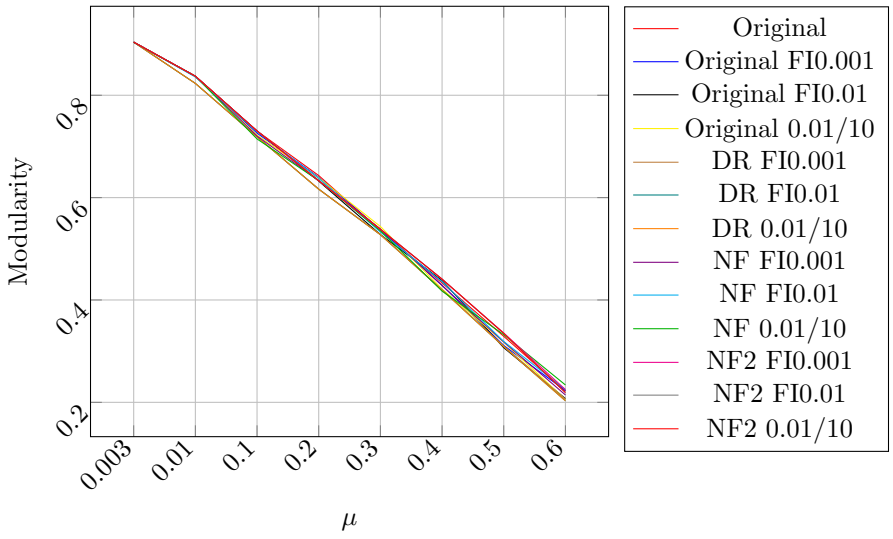


Figure 11.3: NMI

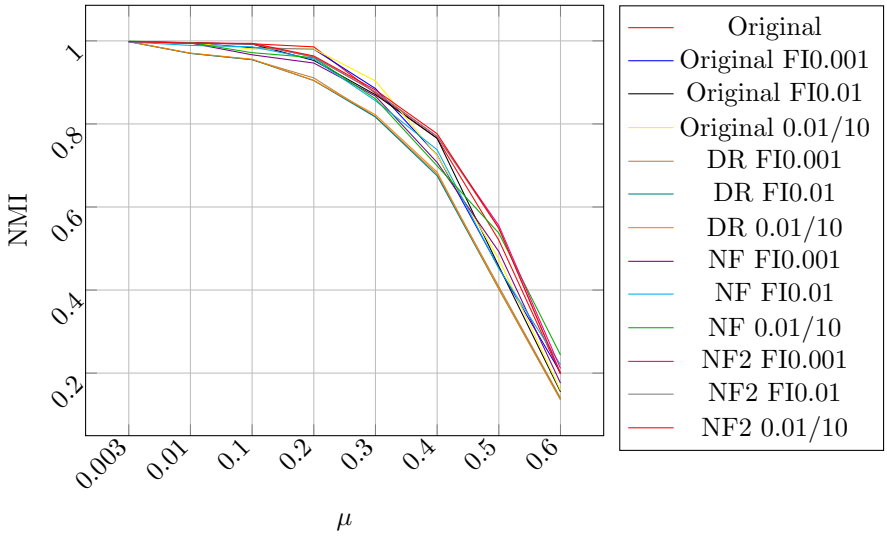
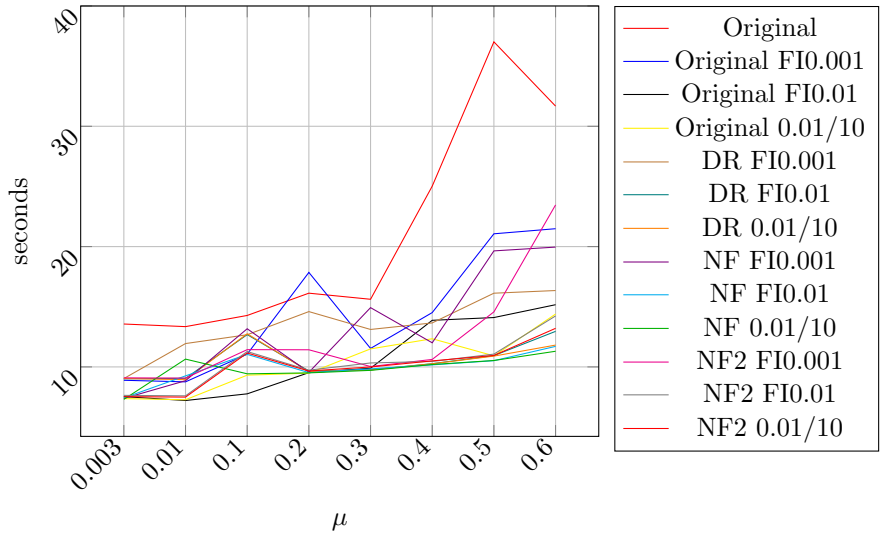


Figure 11.4: Running-time



In Figure 11.2 we can observe that as we increase μ, the modularity sinks. This is not surprising as modularity is measurement of how well defined the community structure is. When we increase μ, the fraction of edges between communities increases. We can also observe that all the tested variants perform

very similarly.

The NMI value plotted in Figure 11.3 also drops significantly as we increase μ. Here we can see bit more difference than in Figure 11.2. This most likely because two different partitionings can have the same modularity, but we are able to decide which of them that are best with NMI. This is because we compare them against the correct partitioning. As NMI measures how far our partitioning is from the correct partitioning, NMI is clearly a better measure when we are comparing the variants. We can observe that all variants follow a similar curve, but the variants using degree ranking performs slightly worse than the rest.

In Figure 11.4, we can observe the time spent on each graph. We can see a trend with increased running-time as we increase μ. The difference in time spent between the variants and the original algorithm seems to be increasing as μ increases.

In Table 11.2, we have the average percentagewise difference compared to the original algorithm over all tested values of μ. The differences between the variants is more visible here. We can observe that the maximum difference in modularity is -3.37%, while the difference in NMI is -10.8%. We can also observe that the neighborhood finder 2 performs better than the original algorithm, while also reducing the running-time. Overall the neighborhood finder 2 performs best in term of modularity, NMI, and running-time. Unsurprisingly, the $FI0.001$ looks lite the threshold variant resulting in the highest modularity, and NMI values. This is unsurprisingly because it is the lowest threshold value we test. All tested variants have an reduced running-time compared to the original algorithm. As with previous tests in Chapter 10, the higher we set the threshold values, the more we reduce the running-time.

Table 11.2: Average percentagewise difference compared to the original algorithm

	Modularity	**NMI**	**Time**
Original FI0.001	-0.88	-2.05	-28.01
Original FI0.01	-1.95	-4.67	-46.54
Original 0.01/10	-1.76	-4.03	-46.17
Degree ranking FI0.001	-3.13	-10.26	-28.87
Degree ranking FI0.01	-3.37	-10.80	-42.88
Degree ranking 0.01/10	-3.37	-10.61	-43.41
NF FI0.001	-1.29	-3.90	-33.52
NF FI0.01	-0.40	-1.45	-46.32
NF 0.01/10	-0.17	1.48	-46.79
NF2 FI0.001	0.47	1.51	-36.75
NF2 FI0.01	0.12	0.62	-45.59
NF2 0.01/10	0.11	0.82	-46.64

11.5 Summary

In this chapter we explained benchmark graphs, and ran our variants on some LFR benchmark graphs. We have shown that the differences between the different variants are clearer when we calculate NMI. Overall the neighborhood finder 2 gave the best partitioning. In the next chapter, we conclude our research into the Louvain-method, and our modifications to it.

Chapter 12

Conclusion

In this thesis we have studied the Louvain-method, a widely used community detection algorithm [5]. We have used this thesis to research two questions raised in the original article presenting the Louvain-method [5]. Firstly, we have researched if the order in which we process nodes have any impact on the running-time or the achieved modularity. We have also studied the impact of introducing thresholds to the Louvain-method. Both questions are important as they not only could improve the Louvain-method, but also other community detection algorithms.

To answer these questions, we have introduced several traversal orderings and several threshold types. We have then tested these variants on both real world networks and benchmark graphs.

Our experiments show that some traversal orderings show some improvement in both running-time, and the achieved modularity. It has also become clear that as the Louvain-method has an estimated running-time of $O(n\log n)$, the time needed to calculate the traversal ordering need to be low. Of our tested traversal orderings, the "Neighborhood finder 2" gave the best results.

In Chapter 9, we introduced thresholds to the Louvain-method. We tested how different types and values of thresholds affected the running-time, and the achieved modularity. Our experiments showed that all the tested thresholds significantly reduce the running-time. The higher we set the threshold, the more the running-time is reduced, but on expense of lower achieved modularity.

12.1 Future work

During our work with this thesis there is several things we would have studied more if we had time.

Can our traversal orderings be beneficial for other community detection algorithms?

Test all of our traversal orderings from Chapter 7 on more benchmark graphs. NMI-seems to give a better comparison than modularity.

All of our traversal orderings are based on graph properties shared by all graphs, like the degree of a node. It would be interesting to use information related to a specific network, to sort the nodes. An example could be if we look at some world spanning networks, we could traverse the nodes by sorting on country.

Study the number of communities returned by the Louvain-method. Can the number of communities tell us something about the running-time?

Create and test even more traversal orderings.

Bibliography

[1] Konstantin Andreev and Harald Räcke. "Balanced Graph Partitioning". In: *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '04. Barcelona, Spain: ACM, 2004, pp. 120–124. ISBN: 1-58113-840-7. DOI: 10.1145/1007912.1007931. URL: http://doi.acm.org/10.1145/1007912.1007931.

[2] Thomas Aynaud. *Louvain-method python implementation*. Feb. 15, 2016. URL: https://bitbucket.org/taynaud/python-louvain.

[3] Vincent Blondel. *Webpage Vincent Blondel /Louvain-method*. 2010. URL: https://perso.uclouvain.be/vincent.blondel/research/louvain.html (visited on 02/13/2017).

[4] Vincent D. Blondel et al. "Fast unfolding of communities in large networks". In: (2008). DOI: 10.1088/1742-5468/2008/10/P10008. eprint: arXiv:0803.0476.

[5] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (2008), P10008. URL: http://stacks.iop.org/1742-5468/2008/i=10/a=P10008.

[6] Ulrik Brandes et al. "On modularity clustering". In: *IEEE transactions on knowledge and data engineering* 20.2 (2008), pp. 172–188. URL: http://ieeexplore.ieee.org/abstract/document/4358966/.

[7] Romain Campigotto, Patricia Conde Céspedes, and Jean-Loup Guillaume. *A Generalized and Adaptive Method for Community Detection*. 2014. arXiv: 1406.2518.

[8] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. "Finding community structure in very large networks". In: *Phys. Rev. E* 70 (6 Dec. 2004), p. 066111. DOI: 10.1103/PhysRevE.70.066111. URL: https://link.aps.org/doi/10.1103/PhysRevE.70.066111.

[9] Wikimedia Commons. *Network of the Zachary Karate Club. Node 1 stands for the instructor, node 34 for the president*. 2017. URL: https://en.wikipedia.org/wiki/Zachary%27s_karate_club#/media/File:Zachary%27s_karate_club.png.

[10] Johan Dahlin and Pontus Svenson. *Ensemble approaches for improving community detection methods*. 2013. arXiv: 1309.0242.

[11] Datasets. *dimacs 10 archive*. 2010. URL: http://www.cc.gatech.edu/dimacs10/archive/clustering.shtml (visited on 04/25/2017).

[12] S. Fortunato and M. Barthelemy. "Resolution limit in community detection". In: *Proceedings of the National Academy of Sciences* 104.1 (Dec. 2006), pp. 36–41. DOI: 10.1073/pnas.0605965104. URL: https://doi.org/10.1073%2Fpnas.0605965104.

[13] Santo Fortunato and Claudio Castellano. "Community Structure in Graphs". In: *Encyclopedia of Complexity and Systems Science*. Springer New York, 2009, pp. 1141–1163. DOI: 10.1007/978-0-387-30440-3_76. URL: https://doi.org/10.1007/978-0-387-30440-3_76.

[14] Santo Fortunato and Darko Hric. "Community detection in networks: A user guide". In: (2016). DOI: 10.1016/j.physrep.2016.09.002. eprint: arXiv:1608.00163.

[15] Santo Fortunato and Andrea Lancichinetti. *LFR benchmark graph generator*. Oct. 29, 2009. URL: https://sites.google.com/site/santofortunato/inthepress2.

[16] Linton C. Freeman. "Centrality in social networks conceptual clarification". In: *Social Networks* 1.3 (Jan. 1978), pp. 215–239. DOI: 10.1016/0378-8733(78)90021-7. URL: https://doi.org/10.1016%2F0378-8733%2878%2990021-7.

[17] M. Girvan and M. E. J. Newman. "Community structure in social and biological networks". In: *Proceedings of the National Academy of Sciences* 99.12 (2002). URL: http://www.pnas.org/content/99/12/7821.

[18] Roger Guimera, Marta Sales-Pardo, and Lus A. Nunes Amaral. "Modularity from fluctuations in random graphs and complex networks". In: *Phys. Rev. E* 70 (2 Aug. 2004), p. 025101. DOI: 10.1103/PhysRevE.70.025101. URL: https://link.aps.org/doi/10.1103/PhysRevE.70.025101.

[19] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. *NetworkX*. May 9, 2017. URL: https://networkx.github.io/.

[20] Joseph Hannigan et al. *Mining for Spatially-Near Communities in Geo-Located Social Networks*. 2013. arXiv: 1309.2900.

[21] Gautier Krings and Vincent D. Blondel. *An upper bound on community size in scalable community detection*. 2011. arXiv: 1103.5569. URL: https://arxiv.org/pdf/1103.5569.pdf.

[22] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. "Benchmark graphs for testing community detection algorithms". In: (2008). DOI: 10.1103/PhysRevE.78.046110. eprint: arXiv:0805.4770.

[23] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. "Benchmark graphs for testing community detection algorithms". In: *Phys. Rev. E* 78 (4 Oct. 2008), p. 046110. DOI: 10.1103/PhysRevE.78.046110. URL: https://link.aps.org/doi/10.1103/PhysRevE.78.046110.

[24] E. Lefebvre and J.-L. Guillaume. *C++ implementation of the Louvain-algorithm*. Feb. 1, 2008. URL: https://sites.google.com/site/findcommunities/.

[25] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[26] David Meunier. "Hierarchical modularity in human brain functional networks". In: *Frontiers in Neuroinformatics* 3 (2009). DOI: 10.3389/neuro.11.037.2009. URL: https://doi.org/10.3389%2Fneuro.11.037.2009.

[27] Rokia Missaoui and Idrissa Sarr. *Social Network Analysis-Community Detection and Evolution*. Springer, 2014.

[28] M. E. J. Newman. "Analysis of weighted networks". In: *Phys. Rev. E* 70 (5 Nov. 2004), p. 056131. DOI: 10.1103/PhysRevE.70.056131. URL: https://link.aps.org/doi/10.1103/PhysRevE.70.056131.

[29] M. E. J. Newman. "Finding community structure in networks using the eigenvectors of matrices". In: (2006). DOI: 10.1103/PhysRevE.74.036104. arXiv: physics/0605087.

[30] M. E. J. Newman. "Modularity and community structure in networks". In: *Proceedings of the National Academy of Sciences* 103.23 (2006). URL: https://www.ncbi.nlm.nih.gov/pubmed/16723398.

[31] M. E. J. Newman and M. Girvan. "Finding and evaluating community structure in networks". In: *Phys. Rev. E* 69 (2 Feb. 2004), p. 026113. DOI: 10.1103/PhysRevE.69.026113. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.026113.

[32] Pascal Pons and Matthieu Latapy. *Computing communities in large networks using random walks (long version)*. 2005. arXiv: physics/0512106.

[33] Josep M. Pujol, Vijay Erramilli, and Pablo Rodriguez. *Divide and Conquer: Partitioning Online Social Networks*. 2009. arXiv: 0905.4918.

[34] Usha Nandini Raghavan, Reka Albert, and Soundar Kumara. "Near linear time algorithm to detect community structures in large-scale networks". In: (2007). DOI: 10.1103/PhysRevE.76.036106. eprint: arXiv:0709.2938.

[35] Joerg Reichardt and Stefan Bornholdt. "Statistical Mechanics of Community Detection". In: (2006). DOI: 10.1103/PhysRevE.74.016110. arXiv: cond-mat/0603718.

[36] Unknown. *Florida Sparse Matrix Collection/Fe_rotor*. 2010. URL: https://www.cise.ufl.edu/research/sparse/matrices/DIMACS10/fe_rotor.html (visited on 04/13/2017).

[37] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*. Vol. 8. Cambridge university press, 1994.

[38] Jierui Xie and Boleslaw K. Szymanski. "Community Detection Using A Neighborhood Strength Driven Label Propagation Algorithm". In: (2011). DOI: 10.1109/NSW.2011.6004645. eprint: arXiv:1105.3264.

[39] Zhao Yang, René Algesheimer, and Claudio J. Tessone. "A Comparative Analysis of Community Detection Algorithms on Artificial Networks". In: *Scientific Reports* 6 (July 2016). DOI: 10.1038/srep30750. URL: https://www.nature.com/articles/srep30750.

[40] Lin Zhang et al. "Subject clustering analysis based on ISI category classification". In: *Journal of Informetrics* 4.2 (Apr. 2010), pp. 185–193. DOI: 10.1016/j.joi.2009.11.005. URL: https://doi.org/10.1016/j.joi.2009.11.005.