Graph Clustering for Long Term Twitter Observations

Community Detection in Incremental Graphs

Aigars Tumanis

Thesis submitted for the degree of
Master in Distributed Systems and Networks
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2021

Graph Clustering for Long Term Twitter Observations

Community Detection in Incremental Graphs

Aigars Tumanis

Graph Clustering for Long Term Twitter Observations

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

In recent years, we have seen that the spreading of Fake News in social media has become a real, potentially life-threatening problem. Discovering and notifying users about misinformation has become increasingly more important. This work attempts to tackle this problem from the point of graph theory, and specifically graph clustering. The objective of this thesis was to create an approach for clustering on-line graphs that grow incrementally by continuously receiving new partitions. Such graphs can appear through the use of scrapers, where data is collected during some time interval.

The main challenge of working with on-line incremental graphs is that they fall into the intersection between on-line and off-line approaches, and as such little to none literature exists on this subject. Through use of an iterative, exploratory, test-driven method several state-of-the-art algorithms were thoroughly tested in a range of different scenarios, with specific intention of using them on on-line incremental graphs. A novel approach to working with these graphs was proposed and a testing framework was designed for this purpose.

The tools presented in this thesis were used to design an algorithm for clustering incremental on-line graphs, called NCLiC. The algorithm was thoroughly evaluated and shows great promise, both in terms of processing speeds and in terms of the clusters it is able to detect. The algorithm is modular in nature and is based on the problem of merging several graphs together without loss of information. The tests have revealed that NCLiC is a far superior approach compared to reclustering the graph, as new partitions are received, and is even competitive with the modern state-of-the-art algorithms, both in terms of runtimes and the discovered clusters.

Contents

List of Figures

List of Tables

Preface

The moment I saw the available spot in the Understanding of Social Media and Digital Wildfires (UMOD) project, working on detection of Fake News in Social Media, I knew I had found the topic for my master's thesis. The spreading misinformation has always interested me and I am extremely thankful to Daniel Schröder for letting be be a part of his project. The way people interact on-line and particularly how information spreads is a topic which is as hard as it is exciting.

While the COVID-19 virus made it harder to meet with each other, the on-line communication channels showed us that we can still meet and interact with each other, in addition to the importance of having and maintaining social connections. I would therefore like to thank Hanne Myklebost Nordengen for being there for me and listening to me babble, even though she had no idea what I was talking about. My supervisor, Johannes Langguth, has been an integral part of the process and I cannot imagine how this would have turned out without his guidance and immense knowledge of the field. I would also like to thank Marius Borge Heir for reading the roughest of drafts of this thesis and offering his feedback on it, but also for taking my mind of it when I needed it the most. And finally, I would like to thank my family and friends for their continued support and words of motivation.

Thank you all!

Chapter 1

Introduction

> — A lie can travel halfway around the world while the truth is still putting on its shoes
>
> *Jonathan Swift [1]*

1.1 Fake News and Digital Wildfires

The quote preceding this chapter has been attributed to many people, among them Winston Churchill, Mark Twain, and John Swift [1, 2]. Regardless of who the original creator was, the quote rang true regarding the way misinformation is spread and the challenges that arise to convince people about the truth, once that has happened. The quote has aged well, and especially today, in the age of social media, the quote has become very appropriate. With the onset of platforms like Facebook, Instagram and Twitter, sharing content has never been easier. Users can easily reach millions of people in just a few seconds and broadcast their opinions and thoughts on past and current events. Social media has led to possibilities that previous generations could not even dream of and brought forth the good, the bad and the ugly sides of ourselves.

In the recent years, online social networks have become increasingly popular as a news distribution platform [3, 4]. For many users, social media has become the main source of news and an important factor in understanding the world around them. While this has the benefit of important news reaching more people faster, it has also opened a door for several new problems. The ease with which anyone can claim to have insight about a specific case or news story has made it easy to deceive people. This can be done through, for example, a professionally looking website or by claiming to be a news outlet. Additionally, Facebook, with its 2 billion users has become a platform for creating echo chambers where conspiracy theories spread. We have seen this happen several times in the past, with the emergence of QAnon, *waging a secret war against elite Satan-worshipping paedophiles in government, business and the media* [5]. Stories about Hillary Clinton's pedophile drug ring in the basement of the Comet Ping Pong fast food restaurant (Pizzagate) [6, 7] and 5G-towers spreading the COVID-19 virus [8] can be spread at an unprecedented rate.

These theories can have real-world consequences, like groups of people destroying and damaging 5G-towers or arriving at fast food chains with shotguns to "rescue the children". A lie today can travel around the world several times before the truth is even conceived.

The increase of misinformation, or fake news, has made the political landscape more segregated and polarised and has become a worldwide problem. This fact has been exploited by companies like Cambridge Analytica, which influenced the behaviour of its targets through the use of targeted campaigns and purposeful misinformation [9]. To battle this, several social media companies have launched counteractive measures, like Twitter flagging potential fake news with an appropriate tag [10], and Facebook deleting groups that spread misinformation [11] . The detection and identification of fake news is a difficult task, but an extremely important one. One way to identify fake news is by studying its spreading patterns and the underlying communication channels; learn where and how fake news originate and how they spread.

1.2 UMOD Project

We have seen the devastating potential that misinformation can have on society. The speed with which this information can be spread makes manual verification of news practically impossible, which leads to the need for an automated misinformation detection system. The Understanding and Monitoring Digital Wildfires (UMOD) project is focused on understanding digital wildfires (fast-spreading, inaccurate, counterfactual, or intentionally misleading information that quickly permeates public consciousness) [12] in order to battle them. While there have been some attempts at creating such systems [13, 14], none of them have been entirely successful. The goal of the UMOD project is to develop a system that can accurately detect misinformation, while also respecting the privacy of users and being open and trustworthy. In order to be able to achieve this, the first step is to investigate the phenomenon of online misinformation by looking at the problem from the standpoints of both computer and social sciences. The project is based on this dual approach, with scientists from both fields cooperating to find a solution to the problem.

In 2019 the researchers at the UMOD project developed the FACT (Framework for Analysis and Capture of Twitter Graphs) framework [3], for gathering the graph structure of follower networks, posts and profiles. The data gathered by the framework serves as the base for further research.

1.3 Topic for the Thesis

This thesis approaches the problem from the computer science perspective. Graph theory can be utilised in order to understand the spreading of misinformation in social networks. The observation of clustering of nodes (community detection) in such networks can be helpful when trying to understand the propagation of information and disinformation.

As this problem is in the intersection of several different fields, amongst them graph theory, natural language processing, and machine learning, the problem can and should be approached from several angles. This thesis will primarily focus on graph theory and especially graph clustering. This approach can provide a benefit of extracting useful information in near-real time, as changes in social media happen quickly. The project can benefit from other approaches, but a good clustering of the underlying graph can be beneficial to other fields and provide them with useful information for further exploration of data.

In the world of social media, with potentially endless amounts of data, the clustering must be done in an efficient manner. This thesis seeks to explore the field of on-line community detection in order to find the best possible approaches, either by using the current state-of-the-art algorithms or creating new ones. The algorithms must be able to work on continuously arriving data, while being fast and producing good clusters. While the motivation for this thesis is rooted in the demands of the UMOD project, the problem is general in its nature and relevant to several problems in graph theory.

Chapter 2

Motivation

2.1 Graph Theory

2.1.1 History

In 1735, the Swiss mathematician Leonard Euler found a solution, or rather lack thereof, for the Königsberg bridge problem - an old puzzle about finding a path over seven bridges without crossing each of them twice. By representing each bridge as an edge in a graph between two points on land and proving that no such path can exist, he made the very first contribution to the field of graph theory. Graph theory is a branch of mathematics concerning graphs, i.e. networks of points which are connected by lines. Much progress has been made since then, and today graph theory is applied to a wide range of different fields, from social science to chemistry [15–17].

2.1.2 Community Detection

A graph is a system of components, defined as $G = (V, E)$, where V is a set of vertices and E is a set of edges between the vertices, showing some form of connection between them [16]. Community detection, also called clustering, is the task of grouping together nodes in groups only using information encoded in the graph topology [15]. This means that one can find nodes that share similarities by only looking at the how the graph is built. An example of this may be the Netflix suggestions, where an algorithm suggests what films a user may like by looking at what other users with similar preferences have liked. The same idea is also used for friend suggestions of Facebook and Twitter, where users with shared connections may be suggested as new potential connections.

Clustering is an essential part of graph theory. This problem has a long history and has appeared in many fields, including computer science, and the first algorithms related to this subject appeared in the early 1970's [15]. With the rise of social media, the field of community detection has become increasingly popular due to the increase of the volume and the variety of data. There are many clustering algorithms available and one of

the goals of this thesis is to present and analyse the current state-of-the-art algorithms in the community detection field.

2.2 Continuously Arriving Data

2.2.1 Data Streaming

Clustering with streamed data (data that is continuously arriving and needs to be processed as soon as it arrives) [18] is not a new problem. Especially in the last decade, several papers have been written on the subject, for example, revolving around counting subgraphs [19] or finding the minimum spanning tree [20]. In comparison to community detection, however, with thousands of published articles and papers on the topic, it is still relatively unexplored.

A data stream can be described as data that is continuously generated by a data source, that sends data in small-sized records simultaneously [18]. The data needs to be processed sequentially on a record-to-record basis, meaning that once a record has been processed and a decision has been made, one can no longer go back and change that decision. Processing of a data stream is a separate field in data science with different demands and constraints related to it.

The FACT framework is a Twitter scraper, that continuously retrieves and processes data about new connections, tweets and retweets. The problem of clustering the data in this format was the origin for this thesis. This problem, however, is not limited to the UMOD project, and the problem was therefore extended to deal with other types of networks as well as the social media networks. However, the Twitter network was still considered as the primary target for the clustering algorithms described in this thesis.

2.2.2 Incremental Graphs

As mentioned previously, there are many algorithms available for clustering static graphs and several of them will be described during the course of this thesis. When working with a data scraper, however, additional measures must be taken. Depending on the amount of information that is being scraped the data may arrive at different intervals. For the domains of social media, and the constantly changing landscape of social interaction, the data will often arrive at a constant rate. When using the static algorithms the processing time would grow in accordance to the scale of the incoming data, as the whole graph would have to be reclustered each time. Eventually, the amount of data would become too large to process regardless of the speed and efficiency of the algorithms. Despite the importance of this problem, it has not yet been explored in the literature, and this thesis is therefore an important step in shining a light on a challenging, but extremely interesting part of the field of graph clustering.

The problem of how to cluster continuously growing graphs, where data arrives in chunks, hereby *incremental graphs*, is the main focus of this

thesis.

2.3 Goals

The main objectives contained within the topic of clustering incremental social media graphs are to find the best method to extract clusters of nodes and to implement on-line and/or off-line clustering on incremental graphs. Additionally, a long-term goal is to use the knowledge acquired to detect changes in online communities and to find possible triggers that can be used to predict these kinds of shifts in the future.

The main goal of this thesis is to find a good way to cluster graphs of incremental nature, as well as to find a good way to test different techniques efficiently. Due to the lack of related work exploring this problem, the thesis is formed as quantitative, exploratory work, with focus on testing different techniques and approaches and to compare them in order to find the best possible solution to the problem.

An additional goal is to present an approach suited for the problem presented in the UMOD project in regards to quality and run time.

2.4 Contribution

This thesis proposes a novel way of working with graphs that grow incrementally, i.e. getting new information in chunks which are larger than edge-wise streaming, making it infeasible to use either static- or streaming-based methods presented in literature today. Several requirements are proposed for working with incremental graphs that are focused on guaranteeing the discovery of good clusters while limiting the use of resources like memory and processing power. Additionally, this thesis presents an approach for clustering incremental graphs and a framework designed for testing different approaches in terms of speed and ability to find good clusters in different incremental graphs.

Chapter 3

Document Outline

The following is an overview of this thesis, including a short description of chapters contained within.

Chapter 1: Introduction presents the background of this thesis and the problem description.

Chapter 2: Motivation presents the motivation for this thesis and introduces incremtental graphs.

Chapter 3: Document Outline provides an overview of this thesis.

Chapter 4: Clustering Theory presents the general theory of graph clustering.

Chapter 5: Community Detection Problems discusses the problems found in graph clustering.

Chapter 6: Modularity-based Methods presents the methods revolving around one of the most popular ways to measure clustering algorithms, i.e. modulairty.

Chapter 7: Other Community Detection Methods presents additional methods used in graph clustering.

Chapter 8: General Properties of Real-World Clusters describes the properties of real-world networks.

Chapter 9: Clustering Algorithms gives a detailed overview of the several algorithms used in this thesis.

Chapter 10: Approach describes the approach used during this thesis.

Chapter 11: Exploration Phase presents the initial exploration and discovery of the tools used in further work done during this thesis.

Chapter 12: Comparing Off-Line Algorithms presents an overview of a selection of current state-of-the-art off-line algorithms.

Chapter 13: Off-Line Algorithms on Incremental Graphs presents performance of off-line algorithms on incremental graphs.

Chapter 14: Comparing Off- and On-Line Algorithms compares performance between an on-line and an off-line algorithm.

Chapter 15: Designing an On-Line Incremental Algorithm presents a first attempt of creating an algorithm for incremental graphs and lessons learned from it.

Chapter 16: Designing a Merging Clustering Algorithm describes the process of creating an algorithm for incremental graph based on lessons learned.

Chapter 17: The NCLiC Algorithm presents a novel clustering algorithm for incremental graphs, NCLiC.

Chapter 18: Evaluation shows the evaluation of the incremental clustering algorithm, NCLiC.

Chapter 19: Large Scale Testing describes and presents tests when running NCLiC on a large set of various graphs.

Chapter 20: Contribution presents the contribution of this thesis.

Chapter 21: Future Work talks about suggestions for improvement of the NCLiC algorithm and future work.

Chapter 22: Conclusion concludes the thesis with final remarks and summary of the thesis.

Chapter 4

Clustering Theory

4.1 What is a Network?

A network, or a graph, is a catalogue of a system's components, often called *nodes* or *vertices*, and interactions between them, called *links* or *edges*. This representation offers a way to describe varying and often complex systems in a way that allows us to compare and study them while filtering out the non-important aspects [16]. Just like Euler represented the bridges in Königsberg as a graph, so can we represent the interaction between proteins, the interaction in social networks or roads between cities in a country and have a common way of working with them. Throughout this thesis, the terms network and graph will be used interchangeably.

4.2 Properties of Networks

When working with graphs, we often use their underlying properties to discover new information about them that is not easily found otherwise. The properties of graphs are a direct result of the properties of the nodes and edges the graphs consists of.

Networks can be *directed* or *undirected*. A network is called directed when all of its edges are directed, meaning that the interaction is one-way. Undirected graphs consist of bi-directional links, meaning that the interaction between nodes works both ways. Additionally, directed graphs can have edges pointing "back", or *back edges*, making them both directed and undirected at the same time [16].

Degree Every node has a degree, that is often denoted as k_i. In an undirected graph, the degree of a node i is its number of edges. The total number of edges(L) in a graph with N nodes can be expressed as

$$L = \frac{1}{2} \sum_{i=1}^{N} k_i \tag{4.1}$$

In a directed graph, however, we distinguish between the *in-degree* of the node and an *out-degree*, where one counts the edges pointing to the

node i and edges pointing from the node i to other nodes separately.

Average degree An important property of a network is its average degree, which is the average of all degrees in a network. For an undirected graph the average degree is given by

$$\langle k \rangle = \frac{1}{N} \sum_{i=1}^{N} k_i = \frac{2L}{N} \tag{4.2}$$

In a directed graph it is important to distinguish between incoming degree, k_i^{in}, and outgoing degree, k_i^{out}. The average degree of a directed network is

$$\left\langle k^{in} \right\rangle = \frac{1}{N} \sum_{i=1}^{N} k_i^{in} = \left\langle k^{out} \right\rangle = \frac{1}{N} \sum_{i=1}^{N} k_i^{out} = \frac{L}{N} \tag{4.3}$$

Degree distribution The degree distribution, p_k, is the probability that a randomly selected node has a degree k. For a network with N nodes the degree distribution is given by

$$p_k = \frac{N_k}{N} \tag{4.4}$$

where N_k is the number of degree-k nodes. One of the reasons for degree distribution being an important metric is that many network properties can be computed from p_k.

4.3 Community detection

Communities, also called clusters or modules, are groups of nodes, or vertices, which share common properties or play similar roles within a graph. Real-world networks often exhibit this community structure, where nodes can be grouped as friends in a social network or proteins with similar structures in a biological interaction network. The purpose of community detection, or clustering, is to detect these community structures by only using the information encoded in the network [15].

Clustering can play an important role in our understanding of the world. By clustering, a network one can find new insights about relationships between nodes in the networks. This can, for example, be used to suggest new products to users or find key actors in social or political networks. The problem of community detection, however, is not trivial and the main elements of the problem itself, like the concepts of community and partition, are not rigorously defined and require some degree of arbitrariness and/or common sense [15].

Chapter 5

Community Detection Problems

5.1 Definition of Community

The main problem in graph clustering is to find a definition of a community. While no definition is universally accepted, and the definition often varies based on the system at hand, intuition dictates that a community is often has more edges "inside" than edges between nodes in different communities. Moreover, communities are often algorithmically defined, i.e. being the final product of an algorithm, without a precise definition. However, the intuitive notion is helpful when looking for communities. From a local perspective, communities are parts of the graph with few ties to the rest of the system. They can be described as separate entities with their own autonomy. From a global perspective, communities are often described as parts of the system that which cannot be removed without affecting the functioning of that system. In order to determine if this is the case it can be useful to see if it is different from a random graph. These graphs are not expected to have a community structure, as the probability of two vertices gaining a link is the same. This notion is used in modularity-based clustering algorithm, and will be described later in this thesis. Additionally, it is natural to assume that similar vertices will often be in the same communities, and there are ways to discover and compare such similarities. Among these are to find their distance from each other in Euclidean space, or to use cosine similarity or the Pearson correlation coefficient between rows and columns of the adjacency matrix [15, 21].

5.2 Graph Partitioning

Graph partitioning is a fundamental issue in clustering and is an integral part of parallel computing, circuit partitioning and design of many serial algorithms. The problem consists of dividing the vertices into groups such that the number of edges lying between the groups is minimal. In order for this method to work, one must specify the number and size of clusters. Specifying the number of clusters is necessary as without it, the optimal

solution would be to leave all nodes in the same cluster. Likewise, without specifying the size of the cluster, the best approach would be to separate the vertices with the lowest degree. Both approaches would yield results that are uninteresting [15].

Most variants of graph partitioning are NP-hard, i.e. as least as hard as an NP-problem (can be solved in polynomial time by a non-deterministic Turing machine), but there are several algorithms that can perform well, despite their solutions not being optimal [22]. Many algorithms in graph partitioning perform a bisection of the graph, iteratively in cases where more than two clusters are desired. Additionally, one often imposes a constraint that partitions are of equal size [15].

Algorithms for graph partitioning are generally not good for community detection, as they need the number of clusters and, in some cases, their size. These are often not known beforehand. It is more desirable to receive that information as the output of the algorithm [15].

5.3 Overlapping Community Detection

Much of the focus within community detection has been on identifying disjoint communities, where a node can only be a member of a single community. It is, however, understood that users in a social network often participate in multiple social groups, like family, friends and colleagues; a researcher can, for example, be active in several areas. Furthermore, the number of communities a user can be a part of is unlimited. This happens in other networks as well, such as biological networks, where a node might perform multiple functions [23]. This notion leads to the fact that networks can have differing overlapping density, which further complicates clustering approaches [23].

Due to its importance, overlapping community detection has gained popularity and several attempts have been made to create good algorithms [15, 23, 24]. Despite the work done in regards to this problem, there are still several unanswered questions that have not been fully addressed. The two most prominent ones are *when to apply overlapping methods* and *how significant the overlapping is*. The question of whether detecting overlapping communities captures any useful information compared to the disjoint clustering is largely unaddressed. The significance of clusters has been explored within the context of disjoint community detection, and can likely be extrapolated to overlapping clustering as well, but no definitive answers have been produced yet [23].

Chapter 6

Modularity-Based Methods

6.1 Modularity

One of the most challenging problems in community detection is to know whether the clusters that have been discovered are "good" or "bad". Many attempts have been made to describe a good cluster, but there is still no formally agreed-upon definition [15]. Nonetheless, there is a consensus that quality functions should be used to describe the performance of a clustering algorithm. A quality function is a function that assigns a score to each partition in a graph. This makes it possible to rank different partitions based on a quality function, where the higher number means better partition. They allow us to compare different algorithms and assess the goodness of their performance.

Modularity of Girvan and Newman [15, 25] is the most popular quality function. It is based on the idea that true community structure in a network corresponds to an arrangement of edges that can be considered surprising. Modularity is given by the number of edges within a group, minus the expected number of edges in a similar graph, placed at random [26]. A random graph is not expected to have a cluster structure, and by comparing the density of edges within communities between a given graph and a random one, a goodness measure can be deduced. The graph to be compared is called a *null model* and is a random graph, but which matches the original in some of its structural features.

It is important to choose a correct null-model. One could demand that the random graph keeps the same amount of edges as the original graph, but this would not be a good representation of the real-wold graphs as the degree distribution would be Poissonian, while in the real-world networks, the degree distribution often follows the Power law. A better null model would be based on the same degree distribution as the original.

By looking at the difference between the densities of internal and external edges of communities, one can detect how well a community is defined. Modularity is defined as follows:

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{w_i w_j}{2m}) \delta(c_i, c_j), \tag{6.1}$$

where A_{ij} represents the weight of the edge between i and j, w_i is the sum of the weights of the edges attached to vertex i. If the graph is unweighted, this would correspond to the degree of that node. c_i is the community to which vertex i is assigned, while the δ-function, $\delta(u, v)$, is 1 if $u = v$ and 0 otherwise and $m = \frac{1}{2}\sum_{i,j} A_{ij}$.

This definition leads to several key properties of modularity [16]:

- The more defined communities are - the higher the modularity score. Meaning that when comparing the modularity scores of two algorithms, one can find the one that performed better by looking at their modularity scores. The algorithm with the highest score would have more clearly defined clusters, resulting in better partitioning, with the highest possible modularity score being 1.
- Modularity can be 0 zero or negative. By placing the given graph into one community, the modularity score is zero. If each node in the given graph is placed into a separate community, the modularity score will become negative.

Figure 6.1 shows the different modularity scores obtained on the same graph. The figure shows optimal partition (a), where communities are clearly separated, suboptimal partition (b), where improvements can be made, 0-modularity (c), where the whole network is a single community, and negative modularity (d), where each node is placed in its own community [16].

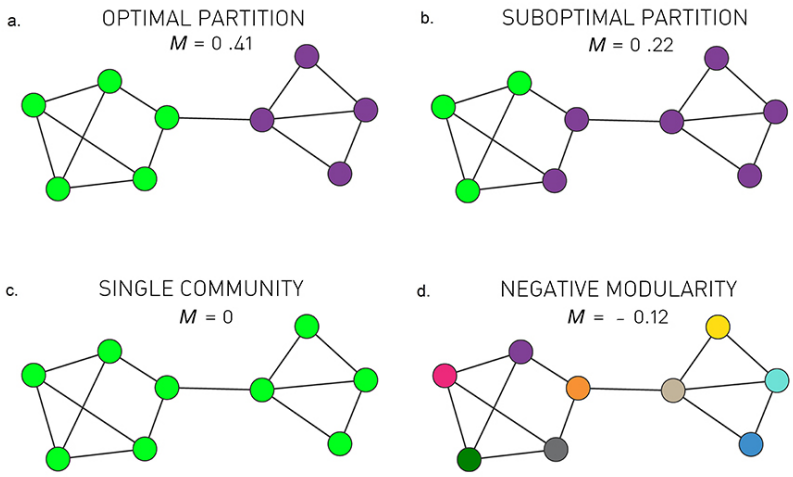


Figure 6.1: *Different modularity scores of the same graph [16]*

Weaknesses While it is generally agreed upon that modularity is a good measure of clustering, one of its main problems is that is has a resolution limit that may prevent it from detecting clusters which are comparatively small compared to the graph as a whole. This is true even when communities are *cliques*, i.e. communities with as many internal links

as possible. This fact has large practical implications since real-world networks often consist of many small communities and a few large ones. Additionally, modularity is highly sensitive to individual connections. Many real-world networks are reconstructed through experiments and surveys and may have edges that should not have been there (false positives). If two small subgraphs happen to be connected with by a few false edges, modularity will put them in the same cluster, which will skew the results of the modularity-based algorithms. Another thing to consider is the null model, which assumes that each vertex can interact with any other vertex. This can be roughly translated to the notion that every vertex knows about every other vertex in the graph. While this may be true for smaller graphs, in huge, real-world networks like the Web and social media graphs this is not the case. Most users will never interact with, or indeed know about, other users that are outside of their social spheres.

Another problem with modularity is that the modularity landscape is characterized by an exponential number of distinct states/partitions, whose modularity values are very close to the global maximum. This means that a modularity based algorithm can return several different community partitions that are extremely different from each other, with no way of knowing whether it actually is the optimal partition or not. Last but not least, modularity-based community detection algorithms do not work on overlapping communities, which exist in real life [15].

6.2 Modularity-optimization methods

6.2.1 Modularity optimization

By definition, a partition with maximum modularity should be the best, or close to it. Despite some caveats with this assumption, due to the range of possible solutions close to that maximum, several algorithms can find fairly good approximations of maximum modularity in a reasonable time [15]. Modularity optimization is the most popular class of methods to detect communities in graphs and the following sections present and describe the main ideas of different approaches to modularity optimization.

6.2.2 Greedy techniques

The first algorithm for maximizing modularity was the greedy method, proposed by Newman [26]. Algorithms that are based on the greedy method are usually quite fast, but the results compared to other algorithms are typically not very good. These algorithms use the agglomerative, hierarchical method where all nodes start in their own clusters, which are then merged based on the highest modularity gain. Once no gain can be achieved, the algorithm stops. One drawback of this method is that the smaller communities tend to be merged into larger ones, even when this is undesirable, due to smaller communities often yielding poor modularity gains [15].

6.2.3 Simulated annealing

Simulated annealing is a probabilistic procedure for global optimization. The algorithms are based on the transition between states, which results in maximized modularity. The states can be defined by "movement"; local, where nodes are moved between clusters and global, where clusters are merged and split. This method can come close to a global maximum, but is quite slow and performs well on graphs with up to 10^4 vertices [15].

6.2.4 Extremal optimization

Extremal optimization was proposed in order to achieve accuracy comparable to simulated annealing, but with significantly higher performance. It is based on the optimization of local variables, expressing the contribution of each unit of the system to the global function. One starts from a partition of the graph into two equally sized groups. At each iteration, the node with the lowest fitness is shifted to the other cluster, and the local fitness on nodes are recalculated. The measure is obtained by dividing the local modularity of the vertex by its degree. The algorithm finds good estimates of the modularity maximum and performs quite well. It may, however, lead to poor results on large networks with many communities [15].

6.2.5 Spectral optimization

Modularity can be optimized using the eigenvalues and eigenvectors of a spectral matrix. By using a modularity matrix, instead of a Laplacian one, we can find clusters in the graph by looking at the elements in the graph and their components. If, for example, the matrix has no positive eigenvalues, the graph has no community structure. Additionally, the values of the eigenvectors are also informative, as they indicate the level of participation of the vertices to their communities, i.e. a vertex with values close to zero are often outliers/border nodes and may be considered to belong to several clusters. The vertices may also be moved in order to gain modularity. The spectral optimization of modularity is quite fast and performs better than extremal optimization, while also being slightly more accurate, especially for large graphs [15].

Chapter 7

Other Community Detection Methods

While modularity optimization is the most popular clustering method, other methods exist. The following chapter presets and briefly describes a selection of such methods.

7.1 Hierarchical Clustering

Many types of graphs, like social networks or protein clustering networks, have a hierarchical structure, i.e. small clusters of nodes are included in larger clusters. In such cases, algorithms that detect the multilevel structure of the graph can be used. Based on the chosen similarity measure, the nodes are put together in clusters, which in turn are combined into larger clusters. Based on whether the new clusters are merged or divided, the hierarchical clustering is split into two categories: *agglomerative* and *divisive* [15].

7.1.1 Agglomerative algorithms

The agglomerative algorithms take the "bottom-up" approach, where clusters are iteratively merged if their similarity is sufficiently high. The result of this approach is a dendrogram, which can be used to discover clusters. Additionally, stopping conditions may be imposed to select a partition or a group of partitions that satisfy a chosen criterion, like a given number of clusters or optimization of some quality function. While hierarchical clustering does not require a preliminary knowledge of the number and the size of clusters, it has some drawbacks. It does not, for example, provide a way to choose the partitions that represent the system best, as the results of this approach are based solely on the similarity measure chosen. Another major weakness of agglomerative clustering is that it does not scale well, making them too slow to use on large graphs [15].

7.1.2 Divisive algorithms

The divisive algorithms are based on the "top-down" approach, where clusters are iteratively split by removing edges connecting vertices with low similarity. The philosophy of divisive algorithms for detection of communities in a graph is to find edges that connect communities and remove them, resulting in only similar nodes being connected [15].

Girvan-Newman Girvan-Newman algorithm [25] is the most popular divisive algorithm. The edges are selected based on the notion of edge centrality, which denotes the importance of edges based on some property or process running on the graph. Girvan-Newman focused on the concept of betweenness, which is a measure of participation of edges in a process. They split it into three types: *edge betweenness*, *random-walk betweenness* and *current-flow betweenness*. Edge betweenness is the number of shortest paths between all vertex pairs. The random-walk betweenness of an edge is given by how likely a random walker is to visit the edge, or the frequency of its visits in a network. Current-flow betweenness is defined by considering the graph as a resistor network with edges having a unit resistance. Comparing these shows that calculating edge betweenness is much faster than both random-walk and current-flow [15].

7.2 Partitional Clustering

Another popular class of methods is the partitional clustering. Given a data set of N points, a partitioning method constructs K ($N \geq K$) partitions of the data, with each partition representing a cluster [27]. The number of clusters is preassigned, and the nodes are embedded in a metric space. The points are assigned to different clusters based on the distance, which represents the dissimilarity between them. The goal is to maximize/minimize a cost function.

K-means clustering The most popular partitioning technique is *k-means clustering*, where the cost function is the total intra-cluster distance, represented by *centroids*, or points representing each cluster [15]. Centroids are real or imaginary locations representing the center of a cluster. The k-means clustering starts by randomly assigning centroids. The nodes are then assigned to the closest centroids, representing different clusters. Once that is done, the centroids are recalculated based on the distance of nodes to the centroid. This process repeats iteratively, updating the position position of centroids until either the *centroids have been stabilised* or *the chosen number of iterations has been achieved* [28].

7.3 Spectral Clustering

All methods and techniques in spectral clustering use eigenvectors of matrices to cluster networks. The methods consist of a transformation

of the initial set of objects, which can be nodes in a graph or data points in some metric space, into a set of points in a space, whose coordinates are elements of eigenvectors. Once that is done, other techniques, like *k-means* clustering can be applied. The main reason for clustering the points obtained through the eigenvectors instead of just using the similarity matrix, is that the eigenvectors make the cluster properties much more evident. By going through the step of finding the eigenvectors, the clustering is easier and additional properties can be discovered, which would not be possible through "regular" means [15].

Spectral properties of a graph can be used to find partitions, for example, by using eigenvectors of the **Laplacian matrix**. Eigenvectors can be combined with other techniques, such as random walks, plotting or conductance to find clustering of graphs.

7.4 Dynamic Clustering

The dynamic clustering algorithms are focused primarily on some form of changing or shifting of circumstances and entails different scenarios, like dynamic features, dynamic data and dynamic clusters. Dynamic clustering algorithms address the challenges that arise when data is non-stationary and changes can occur at various rates. The clustering algorithms can also be applied in situations involving large amounts of data, data streams and incomplete or noisy data [29].

Random walks Algorithms based on random walks can be useful for finding communities. A random walker spends significantly more time inside a community due to the high density of internal edges and the number of paths that can be followed if a graph has a strong community structure. Random walks can be used to define several characteristics, like the distance between nodes, the proximity of node pairs to other nodes in the network, the influence of nodes on other nodes or flow between vertices [15].

Synchronization Synchronization is a phenomenon in which the units of a system are in the same or similar state at every time. This can be used to detect partitions, as the state of nodes in a single community would be the same for those nodes, while differing from the nodes in other communities. The challenging part of these algorithms is to find a good evolution model, such that the state of nodes can be measured. Synchronization algorithms have been shown to give good results in practical examples and even have been used to detect overlapping communities. However, the synchronization-based algorithms may not be reliable when communities are very different in size and further testing is needed [15].

7.5 Statistical Inference

Methods based on statistical inference aim at deducing properties of data sets based on a starting set of observations and model hypotheses. The community detection algorithms using statistical inference try to find the best fit of a model to the given graph, where the model assumes some sort of classification of vertices, based on their connectivity patterns.

Generative models Generative models are often centered around *Bayesian inference* [30], which uses observations to estimate the probability that a given hypothesis is true. It consists of two elements: the evidence one has about the system, and a statistical model for that system. The goal is to determine the parameters of the statistical model in such a way that maximises the posterior distribution of the parameters given the model and the evidence. Bayesian inference is frequently used in the analysis and modelling of real graphs, like social and biological networks [15].

Blockmodeling Block modeling is a common approach in statistics and social network analysis to decompose a graph in classes of vertices with common properties. The reason for this is to obtain a simplified version of a graph. Vertices are usually grouped into classes of equivalence, with two main definitions being *structural equivalence* and *regular equivalence*. Structural equivalence defines vertices as equivalent if they have the same neighbours. Regular equivalence defines equivalence when vertices of a class have similar connection patterns to vertices of the other class (ex. parents/children) [15].

Chapter 8

General Properties of Real-World Clusters

Many papers on clustering present applications on the real systems. Despite the variety of available techniques, in many cases partitions derived from different methods are similar to each other. This may point to the fact that the underlying properties of the graphs are not dependent on the algorithms used.

One of the first addressed issues regarding the properties of real communities was whether there is some special distribution of the size of communities or whether they of roughly the same size. The answer to this was that there seems to be no characteristic size - small communities usually coexist with large ones. Additionally, the sizes of communities seem to follow some form of a power-law distribution [31]. Lescovec et al. [32] showed by using *conductance* [1] that communities are well defined only when they are small in size (about 100 nodes). These communities are often found in the periphery of the network. The other vertices form a big core, in which communities are well connected to each other, making them barely distinguishable.

Looking at graph characteristics, one can look at the roles of vertices and their participation in the network. The types of nodes a specific vertex is connected to may point to its function in the graph. In metabolic networks, for example, the hubs that share most edges with vertices from other clusters are more conserved across species, meaning that they have an evolutionary advantage over other vertices.

Additionally, it has been shown that the degree distribution of the network of communities can be reproduced by assuming that the graph grows according to a simple preferential attachment mechanism, where communities with a large degree have a higher probability of interacting/overlapping with new communities [33].

[1] Ratio between the cut size of the cluster and the minimum between the total degree of the cluster and that of the rest of the graph

8.1 Applications on real-world networks

Fortunato [15] describes that some characteristics are shared between real-world networks. These are that they often display inhomogeneities, or variations, and reveal a high level of order and organization. The degree distribution in such networks is broad and follows a power law, where few nodes have many connections, while most of the nodes only have a few links to other nodes. The degree distribution is also inhomogeneous not only globally, but locally, meaning the high concentration of edges within groups of vertices and low concentration of edges between groups. In community detection, this is called community structure, and most of the algorithms use this fact to detect clusters. Real-world networks often display hierarchical organization, with clusters being comprised of smaller clusters, which again are comprised of even smaller clusters.

The ultimate goal of clustering algorithms is to find properties of and relationships between nodes that are otherwise not visible by direct observation. While most of the work in the field has been directed towards finding new algorithms, several works have had a goal of understanding real systems.

Networks based on social interaction between people have been studied for decades. However, with the emergence of social media and other online forms of communication, it has been increasingly easier to gather large amounts of reliable data to work with. This has lead to some interesting observations. The study of the Belgian telephone communication [34] found the linguistic split of the Belgian population by using clustering. A study of e-mail communications between the employees of HP Labs [35] detected clusters that clearly matched the organizational structure of departments and project groups. A study of Facebook friendships in different American universities [36] found that communities were organized by class year or by house affiliation, depending on the university. Other similar experiments have been conducted, showing that clustering algorithms could detect real-life patterns of communication and interaction.

Chapter 9

Clustering Algorithms

9.1 Infomap

9.1.1 Overview

Rosvall et al. proposed *map equation* [37] algorithm in 2009. The main motivation for the Infomap algorithm was the fact that the most popular community detection algorithms, such as the modularity-based ones, infer module assignments based on the underlying network formation process. Rosvall proposed looking at networks as structures for transmitting information, therefore carrying a flow, and using that information to detect communities in a network.

Rosvall proposed that networks are characterised as integrated systems where different components have inter-dependent interactions with each other, i.e. components A and B have an influence on the components B and C and so on. These types of networks often carry some kind of flow that describes the behaviour of different components of the system - passengers travelling between airports, money transferred between banks and information exchanged between friends. The composition of the network dictates this flow and the paper presents the map equation that utilises this fact. By using the trajectory of a random walker [15] on the network, we can map the structure of the network. The reason for using the random walker is that while some networks can have trajectories of movement, most of them lack this kind of information and the best we can do is predict this information, based on the possible trajectories the network structure allows for.

To map the flow, we need to use some code assignment of the flow. The most efficient way to do this is to use Huffman code [38] - a method for data compression that uses a binary tree to assign shorter codes to more frequent nodes and longer codes to less frequent ones. In order to maximise the efficiency of this, the authors propose to use modules for regions of the graph where the random walker often spends time in. These regions can be assigned their own codewords, which can make the path-map even smaller. These modules, in turn, contain other smaller modules and so on.

The Infomap algorithm can be used on both undirected graphs and directed graphs. The difference between the two approaches is the

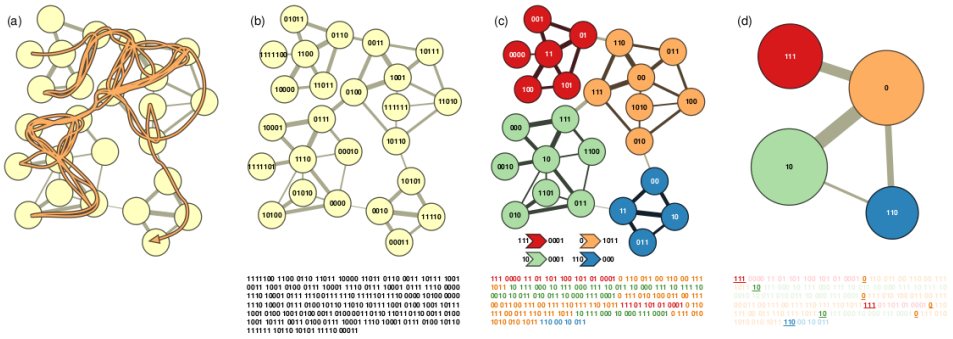


Figure 9.1: *Huffman Codes used in Infomap [37]*

introduction of a small teleportation probability which transforms random walker into a random surfer. This resembles the main idea behind the PageRank algorithm [39], proposed by Page and Brin (1999). Additionally, the algorithm can be used on weighted, as well as unweighted graphs. In case of weighted graphs, the weights represent the probability of the random-walker to follow an edge.

9.1.2 Algorithm

The core algorithm is presented as follows: neighbouring nodes are joined into modules, which are subsequently joined into supermodules and so on.

1. Each node is assigned to its own module.
2. Each node is moved to the neighbouring module that decreases the map equation the most. If no decrease in achieved, the node is not moved.
3. Step 2 is repeated each time in a new random sequential order until no move generates a decrease in the map equation.
4. The network is rebuilt with modules in the last level forming the nodes at the current level. Nodes are again joined into modules. This rebuilding is repeated until the map equation cannot be reduced further.

The core algorithm may be improved by either using *submodule movement*, which allows the main algorithm to be applied to submodules or *single-node movement*, which allows for the main algorithm to be reapplied to individual nodes after their placement into modules. The extensions can be applied together sequentially as long as the clustering is improved. Since the algorithm is stochastic, it can be reapplied any number of times, as long as there are gains to be achieved.

Rosvall et al. concludes the paper with the notion about the difference between modularity- and flow-based approaches. They show how flow-based algorithms differ from modularity based ones by presenting two graphs generated from the same underlying network, with the only

difference being the direction of links. From the perspective of modularity, the graphs are identical, while from a flow-based perspective, they are completely different. They conclude that depending on the goal, different algorithms may be used - if one was interested in analysing how the networks were formed modularity may be better. However, if the flow and dynamics of the network are more important, the flow-based approaches may be preferred.

9.2 Louvain Method

9.2.1 Introduction

The Louvain Method (LM) [34] was developed and presented by Blondel et al in 2008. The algorithm utilises the fact that modularity can be used as an objective function to optimize the clustering of the graph. They proposed a method for extraction of the community structure of large networks based on modularity optimization. Due to this problem being computationally hard, however, different approximation algorithms were necessary when dealing with large networks. LM uses the modularity optimization method proposed by Clauset et al. [31] with additional enhancements in order to circumvent the deficiencies the method has. The proposed algorithm consists of recurrently merging communities such that the resulting modularity is maximized.

Until the introduction of the algorithm, the largest network that had been processed consisted of around 5 million nodes. Using the Louvain Method (LM), a network of 118 million nodes was processed successfully. The processing took about 152 minutes, which was unprecedented at the time.

9.2.2 Algorithm

The algorithm finds high modularity partitions of large networks while producing a complete hierarchical community structure for the network - resulting in different resolutions of community detection. The algorithm is divided into two phases that are repeated iteratively. Figure 9.2 depicts the steps of the algorithm, where each step is comprised of two phases: first, modularity is optimized by changing of communities locally, and second, where discovered communities are aggregated in order to build a new network of communities.

First phase In a community with N nodes, every node starts in its own community. For each node pair (i, j), the gain of modularity is evaluated, and the node is placed in the community that maximizes modularity. If no positive gain is achieved, i is kept in its original community. This process is applied until no additional modularity gain is possible. When this happens, the first phase is complete.

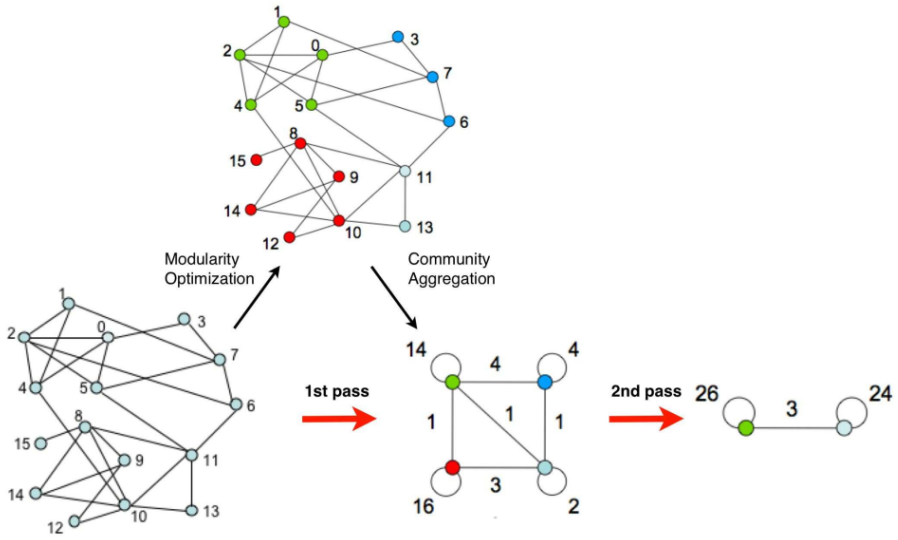


Figure 9.2: *Steps in Louvain algorithm [34]*

Second phase The second phase consists of building a new network whose nodes are now the communities found during the first phase. Once this is done, it is possible to reapply the first phase to the newly created network.

Passes The first and second phases are repeated iteratively, and most of the computing happens during the first pass, as during the future passes the number of meta-communities is decreased. The passes are iterated until no more changes can be done and maximal modularity has been reached. It is important to mention that the optimum discovered is not a global, but a local one.

9.2.3 Performance

Blondel et al. reported that the Louvain Method was more effective than other algorithms at that time, where the network size limits were put due to limited storage capacity rather than limited computation time. Tests had shown that on a typical, sparse graphs, the algorithm runs in a near-linear time, since modularity computation is fast. Additionally, the algorithm gets faster in later iterations, since communities get merged and less computations are needed. In fact, the most intensive computation is done during the first iteration. The resolution limit problem, which shows that modularity based algorithms merge communities that are smaller than a given size, seems to be circumvented. The authors of the paper express that intermediate results may be used to monitor how smaller communities have been merged into larger ones.

The performance can be further improved by using some simple heuristics, like stopping the first phase of the algorithm when the gain in modularity is below a threshold or by removing nodes of degree 1 (leaves)

from the original network and adding them back after the community computation.

9.3 Leiden Algorithm

9.3.1 Overview

According to the paper by Traag et al. [40], the Louvain algorithm has a major defect which may result in it yielding badly connected communities. The authors proposed an update of the Louvain algorithm, which they called *Leiden algorithm*. It promises that the proposed algorithm yields communities that are guaranteed to be connected. It should also run faster than the Louvain algorithm.

According to the authors, the Louvain algorithm may yield disconnected communities due to the fact that nodes can be moved from one community to another at any time. The moved node may have acted as a bridge between two communities. Moving those nodes may lead to these communities becoming disconnected. When this happens, the community should be split up, but the algorithm does not take this into account, and the result becomes sub-optimal. Additionally, the Louvain algorithm may detect communities that are weakly connected. Thus, in general, Louvain may find arbitrarily badly connected communities. The authors note that this problem is different from the resolution limit, which states that smaller communities may be "hidden" in larger communities, regardless of the underlying structure. This problem persists in several variations of the algorithm. Moreover, the problem seems to be aggravated by iteration, since the algorithm has no mechanism for fixing the disconnected communities that appear in the previous iterations.

9.3.2 Algorithm

In order to fix the problems with Louvain, the authors propose a new algorithm, which guarantees that the communities are well connected. The algorithm makes use of the smart local move [41], fast local move [42, 43] and random neighbour move [44], which use heuristics for moving nodes between clusters in a novel way, improving the modularity score and speed of the algorithm. Figure 9.3 shows the steps of the algorithm on two levels, the first level considers the nodes of the graph, while the second one considers the discovered communities from the first step. The Leiden algorithm consists of three phases:

1. Local movement of nodes (a - b)
2. Refinement of the partition (c)
3. Aggregation of the network based on the refined partition (d - f)

In short, the Leiden algorithm starts from a singleton partition and then moves nodes from one partition to another. These partitions are

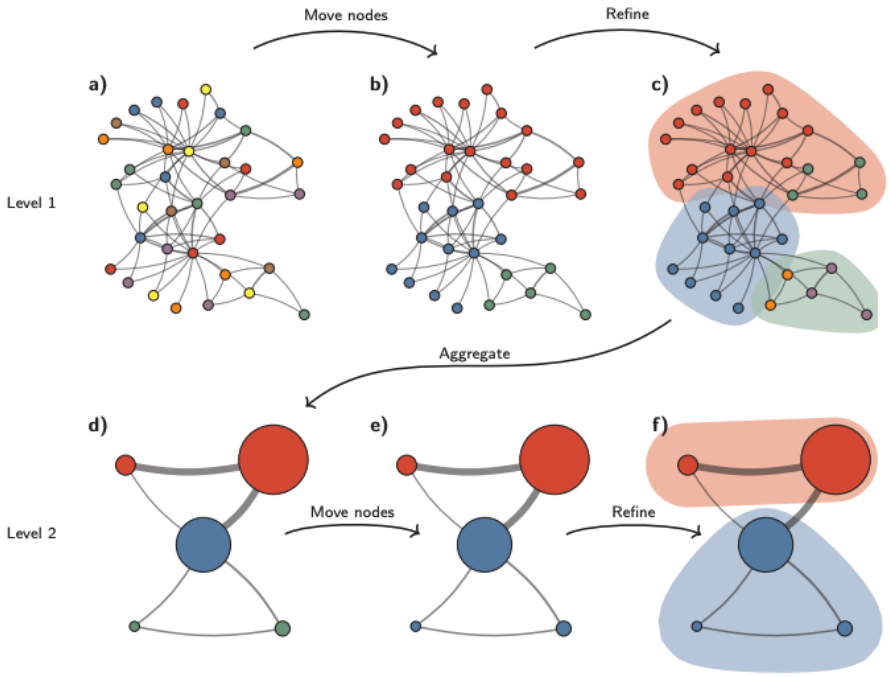


Figure 9.3: *Steps in Leiden algorithm [40]*

then refined. Based on the refined partitions, an aggregate network is created which again is used as the base for movement and refinement of communities. These steps are repeated until no further improvements can be made.

In the Louvain algorithm, an aggregate network is created based on the partition p resulting from the local moving phase. The idea of the refinement phase of the Leiden algorithm is to identify a partition $p_{refined}$ where p may be split up into separate communities. The aggregated network is based on $p_{refined}$, in which it is obtained in the same way communities are obtained, i.e. every node starts in the same community and nodes are merged into communities based on their connectivity, which leads to an increase of their quality function. The way these communities are merged is chosen based on precomputed probability, as long as the quality function is increased.

Another important difference between Leiden and Louvain is the implementation of the local moving phase, where Leiden uses the fast local move procedure. While Louvain keeps revisiting the nodes until no more gain can be discovered, in Leiden, only the nodes whose neighbourhoods have changed are visited. The fast local move procedure keeps a queue of nodes which should be moved, and when a change in a neighbourhood occurs, the nodes in that neighbourhood are added to the queue and the process is repeated until the queue is empty. This is an improvement of the Louvain algorithm, where nodes are being revisited, independently of whether they have been moved or not.

The algorithm provides several guarantees; in particular, that it yields

communities that are guaranteed to be connected. Additionally, when the algorithm is applied iteratively, it converges to a partition in which all subsets of all communities are locally optimally assigned, meaning that no subset can be moved to a different community.

The paper shows that when comparing the two algorithms, the number of badly connected communities was higher for Leiden after the first iteration. However, after only one additional iteration, this number drastically improved. This was not the case for Louvain, where the number of badly connected clusters usually increased. Furthermore, Leiden proved to be able to handle larger networks when dealing with difficult partitions of large synthetic networks generated with high mixing parameter (probability of creating edges between nodes). For higher values of the mixing parameter, Leiden outperformed Louvain by 10 - 100 times for the largest networks. In regards to the real-life networks, the Leiden algorithm was reported to be 2 - 20 times faster than Louvain.

9.4 SCoDA

9.4.1 Overview

The three aforementioned algorithms vary in the way they work and have their strengths and weaknesses. The main challenge for these, and most of the other clustering algorithms is to scale to the sizes of real-world networks. Alexandre Hollocou et al. proposed in 2017 a community detection algorithm that is based on streams of edges, runs in linear time and scales to large networks. The proposed algorithm is called Streaming Community Detection Algorithm (SCoDA) [45] and is based on the observation that if an edge is picked randomly, this edge is more likely to connect two nodes in the same community than two nodes in different communities. The main advantage of SCoDA is that it can be run on networks of sizes that would be prohibitive for most off-line state-of-the-art community detection algorithms.

Most of the commonly used community detection algorithms are based on some kind of maximization of a quality function (often modularity) or other methods, such as random walks, spectral clustering, clique percolation, statistical inference or matrix factorization. Some algorithms designed for clustering streamed data have also been proposed, such as counting subgraphs [19], computing matchings [46], finding the minimum spanning tree [20] or graph sparsification [47]. While many of these approaches find good clusters in graphs, they are computationally intensive, time-consuming and fail to scale to the sizes of modern real-life networks.

According to the authors, the SCoDA algorithm has linear execution time and a low memory footprint as it only stores two integers per node and processes each edge only once, making it viable for processing huge graphs of sizes that would be prohibitive for other algorithms.

9.4.2 Algorithm

The algorithm uses a streaming approach by processing each edge in an edge-list separately. Each edge is "streamed" in a random order, by shuffling the edge list beforehand.

For each arriving edge (u, v) the algorithm places u and v in the same community if the edge arrives early and splits the nodes into separate communities otherwise. In this algorithm, the notion of *early* is that the current degree of nodes u and v, based on previous edges, is low.

Steps of the algorithm:

1. The degree threshold, D, is chosen based on the degree distribution of the network
2. shuffle the list (generate a random permutation)
3. place each node is in its own community
4. for each new edge; u joins v if its degree is lower and vice versa. If the degree of one of them is bigger than a given degree threshold (D), no action is taken

The degree threshold (D) is the only parameter of the SCoDA algorithm. According to Hollocou et al., choosing the correct input for this parameter is critical, as choosing the wrong D can produce significantly worse results. The authors claim that the quality of the clustering can degrade by about 50 % if the wrong threshold variable is chosen. They propose using the *mode* of the degree distribution, i.e. the degree that appears most often in the graph, excluding the leaf nodes. The authors claim and show that calculating degree mode takes linear time and produces the best results for nearly all graphs. In regards to performance, the papers claims that SCoDA is 10 times faster than the state-of-the-art algorithms, such as Louvain [34] and SCD [48] (not covered in this paper), and shows better detection scores on large graphs.

Chapter 10

Approach

10.1 Introduction

The Understanding and Monitoring Digital Wildfires (UMOD) project is focused on understanding the origins and spreading of digital wildfires, as previously mentioned. In order to achieve this goal, clustering techniques can be applied, as seen in the FACT paper [3]. Due to the non-typical data flow of the FACT framework, however, the clustering methods currently available may be sub-optimal to achieve that goal. The following thesis explores possible approaches more suitable for this type of problem. The main focus of this thesis is, therefore, to find the best approach for discovering clusters in Twitter networks enabling long-term observations. This was done by looking at the most promising off- and on-line clustering algorithms, presented in Chapter 3. The algorithms were then tested and evaluated by their performance and ability to detect clusters in different scenarios. The findings were then used to design a new algorithm that worked for the problem.

10.2 Method

This thesis is based on an exploratory, iterative analysis. The preliminary exploration of literature seems to indicate that there is little to no research on this subject. Due to this fact, the most important contributions of this thesis will be the presentation and description of tests and analysis of results discovered under way. A detailed explanation of the tests conducted will be presented in each chapter, and the key findings reported and analysed. These findings will be presented in an iterative fashion, where the discoveries from previous iterations will present new challenges and possibilities. The arising questions and problems will lay the foundation for the next iterations. This iterative way of working will conclude with a proposal of a possible approach to cluster incremental graphs in a way that results in good partitions of clusters with a special focus on runtime.

The method used in the thesis is a mix of qualitative and quantitative analysis, where the qualitative analysis serves as the base for quantitative

analysis. Quantitative analysis can be counted, measured and explained using data, while qualitative analysis is more descriptive and conceptual, and can be categorised based on traits and characteristics [49]. For this thesis, different algorithms will be presented and analysed while looking at their underlying characteristics and concepts. When analysing different approaches and algorithms - their strengths and weaknesses - qualitative analysis will be used. Each algorithm will to be looked at an individual level but also through the lens of other state-of-the-art algorithms. However, during the analysis of the performance of different algorithms, quantitative analysis will be utilised. Additionally, new information can be gained through testing. In order to achieve that, the results need to be presented in a clear way that can be used to discover new possible actions. The results must, therefore, be categorised with care and precision.

The exploratory approach and the use of qualitative analysis will lead to a deeper, more thorough quantitative analysis. Each step in this process will produce more information that will be used in later iterations, concluding with a presentation of the final discoveries and contributions of this paper.

10.3 Evaluation Criteria

When working with community detection, it is important to find algorithms with a good balance between the quality of the clusters it discovers on average, and the time it takes to run it. An algorithm that can find perfect clusters will not be usable if its runtime exceeds what is reasonable on the hardware that is available today. An example of this is the NP-hardness of modularity - while highly useful characteristic, it may not be run in polynomial time in its pure form. Any efficient modularity optimisation algorithms are therefore heuristics [50]. The opposite is also true - an algorithm that is blazingly fast, but detects poor clusters is unsuitable for its purpose. Therefore, it is important to find a good balance between runtime and performance. The goal is to find the best possible clusters in the shortest amount of time. Since modern-day social media graphs are nearly boundless in size, a limit on the runtime must be set.

Throughout this thesis, two measures will be used. Firstly, **time**, including asymptotic running time, to measure the speed of the algorithms and secondly, **modularity**, to measure the performance and the ability to find good clusters. When measuring the modularity of algorithms, we must have something to compare them to. Only presenting a modularity score on its own will not tell us anything useful. Due to the differing characteristics of networks, different graphs may have varying maximum modularity scores. Comparing two graphs that have widely different maximum attainable modularity scores to each other would be counterproductive. The same is true for runtime, as presenting the performance of the algorithm in seconds, minutes and hours would quickly become outdated as better hardware is introduced. A baseline test must be

established before proceeding to designing and implementing new types of algorithms.

10.3.1 Asymptotic Time Complexity

When talking about runtimes of algorithms, it is often counterproductive to use actual times, as the same algorithms may perform differently on different hardware setups. Additionally, as hardware gets better, the performance of the slowest algorithms will get faster. Using asymptotic time complexity alleviates these problems [51]. The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms in order to be able to compare them without the need for implementation and running of the algorithms on actual machines. The asymptotic notation is used to measure and compare the worst-case scenarios of algorithms [52]. The following three asymptotic notations are often used to represent the time complexity for a given algorithm $g(n)$ [53]:

Θ Notation: a theta notation bounds a function from above and below, defining exact asymptotic behaviour of an algorithm.

$$\Theta(g(n)) = f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ s.t:}$$
$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n \geq n_0$$

O Notation: a way to express the upper bound of the running time of an algorithm, i.e. it bounds the function only from above.

$$\mathrm{O}(g(n)) = f(n) : \text{there exist positive constants } c, \text{ and } n_0 \text{ s.t:}$$
$$0 \leq f(n) \leq c * g(n) \text{ for all } n \geq n_0$$

Ω Notation: is the least used asymptotic notation. It provides an asymptotic lower bound on a function.

$$\Omega(g(n)) = f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t:}$$
$$0 \leq c * g(n) \leq f(n) \text{ for all } n \geq n_0$$

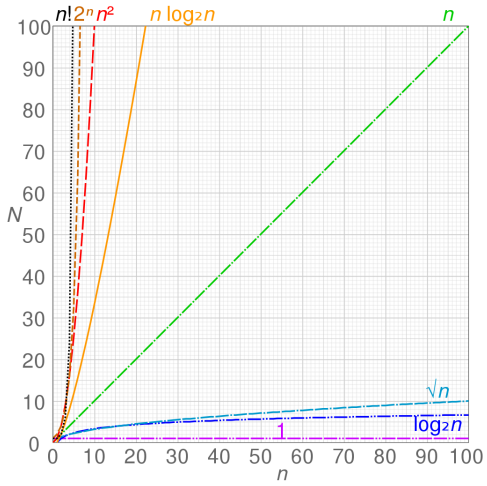


Figure 10.1: *Comparison of asymptotic running times [54]*

Figure 10.1 shows the relationship between often used asymptotic annotations. The figure shows the growth of a function N, as the input size n increases. The lower the increase of N, the better. Example of this is when the asymptotic time complexity is said to be $O(1)$, the runtime would be the same, no matter the input. This, however, is not the case in real-life, and linear growth ($O(n)$) is often the best possible running time one can achieve.

That said, when comparing different algorithms it can sometimes be hard to calculate their asymptotic runtimes. It can therefore be useful to compare the actual running times, as one can present the times as relations between each other. For example, saying that one algorithm is four times faster than another may be a good indication of their comparative performance, especially when they have the same asymptotic running time. This is the reason for including actual running times in this thesis.

10.3.2 Modularity

In order to be able to conduct good evaluations of different algorithms, a good way of measuring their ability to detect communities is needed. A possible approach is to restrict the testing of algorithms to networks with known ground truths, such as Zachary's Karate Club [55] or ground truth graphs from the Stanford Large Network Dataset Collection (SNAP) [56]. However, the number of such graphs available is limited. Furthermore, many ground-truths are based on assumptions and are often extrapolated from smaller sample sizes, so the clusters presented in these network sets may not reflect the actual real-world clusters. An example of this is the email-Eu-core network [57] from SNAP, which is retrieved from a large EU institution. One of the apparent problems with this is that having an e-mail from one department does not guarantee that you are a part of that department in real life. The person could have changed departments, but not the e-mail, or could be working on something unrelated to their department, but still using the e-mail that they have received. While

high recall and precision are important for any clustering algorithm, the incremental graphs set additional limitations on speed and memory. While not entirely a strict streaming scenario, where the algorithm cannot go back and redo decisions, the number of such actions should be limited to a minimum as they are often resource-heavy. While other techniques were considered, such as Normalized Mutual Information Score (NMI) [58], modularity is the most used and trusted technique, despite its weaknesses. It was therefore chosen as the primary way of measuring "goodness" of the algorithms in this thesis.

10.4 On-line versus Off-line Algorithms

Throughout this thesis the terms **on-line** and **off-line** algorithms will be used. The terms are coined by Richard Karp in 1992 in *On-Line Algorithms versus Off-Line Algorithms: How Much is it Worth to Know the Future* [59]. An on-line algorithm is an algorithm that receives a sequence of requests and performs an immediate action in response to each one. The off-line algorithm, on the other hand, performs actions once all requests have been made. It may then change the sequence of execution steps based on the sequence of input, or as Karp put it: *"the off-line algorithm knows the future"*. The inherent characteristic of social media graphs, including Twitter, with data appearing continuously makes on-line types of algorithms more suitable than their off-line counterparts. When measuring the effectiveness of the the on-line algorithm Karp argues that one can compare it to the worst-case scenario of its off-line counterpart. In this thesis, however, the comparison of the on-line algorithm will be made with the average scores of the best-performing state-of-the-art off-line algorithm available. While tremendously harder to achieve good results, if such results are possible, the contribution of this thesis would be even more valuable.

10.5 Benchmark Networks

Throughout this thesis, many different graphs and networks are used. Some of the graphs are used on several occasions, in order to establish a baseline and to be able to compare different algorithms. The most used ones are:

Zachary's Karate Club Zachary's Karate Club [55, 60] is well known in network science. It is based on a real-life karate club that split right after the information about the graph had been gathered. The way the club split makes it appropriate for clustering problems, as it is known how and why the club split. Unfortunately, the preliminary tests have revealed that the modularity based algorithms often cluster the graph into three instead of two clusters, as it is known to have happened. The graph is small, leading to clustering algorithms running fast, independent of their

potential inefficiency, which makes the graph good for preliminary tests. The graph consists of 34 nodes and 78 edges and is presented in Figure 10.2

A Stochastic Block Model-generated graph When measuring the goodness of clustering algorithms, it is often difficult to assess their qualities when the truth is not known. This is the reason for generating graphs, where you have control over all the data embedded in them. One of the ways to generate a graph is by using the stochastic block model. The same graph has been used throughout this thesis - it resembles clusters that may appear in a social network, with a high count of edges between nodes in a cluster and low count of edges between nodes in different clusters. The way it is generated makes it relatively easy to cluster compared to many real-life graphs, and also makes it quite fast to run clustering algorithms on.

For testing purposes, a graph of 1100 nodes and 4 063 edges was created by using the NetworkX Python library [61]. It consists of 7 communities varying in size from 50 to 300 nodes[1]. The graph is depicted in Figure 10.3.

SNAP Twitter Graph The Stanford University Network Database [62] has a large collection of real-world graphs. The graph was chosen due to the fact that it is scraped from Twitter, which makes it highly relevant for this thesis. Additionally, due to its size of over 80 000 nodes and over 1.7 million edges, it is large enough to test for and potentially discard algorithms with prohibitively high running times. It was also deemed to be a good representation of real-world social media graphs. The graph is depicted in Figure 10.4

DIMACS10 The SuiteSpace Matrix Collection [63] is a widely used set of sparse matrix benchmarks collected from a wide range of applications. The DIMACS10 set [64] was presented in the 10th Discrete Mathematics and Theoretical Computer Science (DIMACS) Implementation Challenge, where the main topic was graph partitioning and graph clustering. The main goal of DIMACS is stated as follows *"DIMACS Implementation Challenges address questions of determining realistic algorithm performance where worst-case analysis is overly pessimistic and probabilistic models are too unrealistic: experimentation can provide guides to realistic algorithm performance where analysis fails"*. The graphs consist of both real-world and synthetic networks. Furthermore, they are split into partitioning problems and clustering problems. These graphs are highly appropriate for this thesis.

The set consists of 151 graphs varying in size and complexity. The size varies between 340 and 265 million edges. All graphs are undirected, making them suitable for computing modularity.

[1]Number of nodes in each community: 300, 200, 200, 150, 100, 100, 50

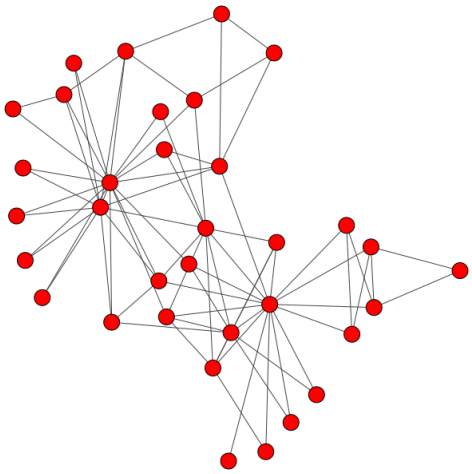


Figure 10.2: *Zachary's Karate Club*

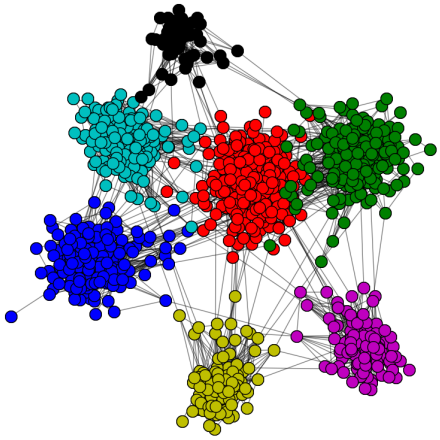


Figure 10.3: *Stochastic Block Model-generated graph*

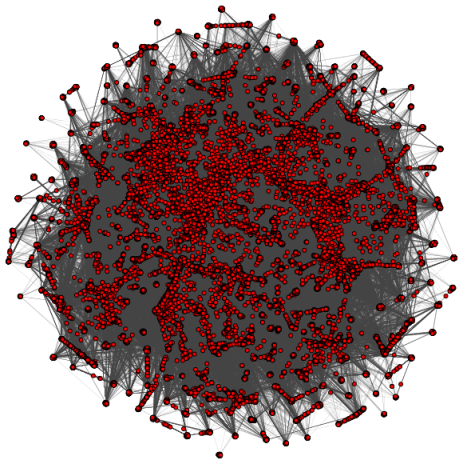


Figure 10.4: *SNAP Twitter Graph*

Chapter 11

Exploration Phase

11.1 Algorithms

In order to get an overview of the currently existing clustering algorithms, the work on this thesis started by conducting an exploration phase. During this phase, different algorithms and their implementations were tested. This was done in order to find the best algorithms to be used in further research. Due to the fact that many algorithm vary in performance and quality of detected clusters, the first step of this thesis was to gather information about state-of-the-art algorithms and perform tests on the ones deemed to fit the problem the best. The next step was to find the best implementations of these algorithms and test them against each other.

During the exploration phase of this thesis, many different techniques and algorithms were tested and several different tools were found. The algorithms were tested for speed and modularity score, in addition to ease of use and ability to be modified. Several research papers were also explored to find the most promising algorithms for the problem at hand. Santo Fortunato's survey [15] was extremely useful starting point for this. The following algorithms were tested during this period:

- K-core: clusters networks by pruning nodes with degrees less than k recursively [65].
- K-clique: discovers cliques (groups sharing k-1 neighbours) using the percolation method [66].
- K-means: uses the notion of centroids to discover clusters [67].
- Greedy modularity: discovers communities by placing each node in their own community, then merging pairs of nodes that maximise the modularity the most [26].
- Fast-greedy modularity: an improved version of the greedy modularity approach that can be run on large graphs [68].
- Edge-betweenness based clustering: clustering based on the number of shortest paths going through a node, which is often correlated with the "importance" of a node in a network [25].

- Louvain method: community detection method based on modularity optimisation, where each node starts in its own community and then merged or moved to a different one as long as the modularity score increases [34].
- Leiden method: an improved version of the Louvain method, that provides guarantees about the goodness of detected communities [40].
- Infomap: community detection method based on the idea that a random walker would visit the most important nodes more often, than the non-important ones [37].
- SCoDA: streaming community detection algorithm that is based on the order that the edges arrive in and the likelihood that the incoming edges are likely to connect nodes that are a part of the same cluster [45].

11.2 Tools and frameworks

Throughout this thesis, several different tools were used. Python was used as the primary coding language. While it is not the fastest, it is easy to create prototypes in. Additionally, it has a large data science community, meaning that many different packages and tools are available. Since this thesis is of an exploratory nature, the deficiencies in processing speed of Python in comparison to languages like Java or C/C++ are not critical for the end result.

Other tools have also been used, like Gephi [69], a graph analysis and plotting software, and PyCharm [70], a Python IDE. Furthermore, several different Python packages were tested, among them NumPy [71], NetworkX [61], Python-based version of igraph (Python-igraph) [72] and NetworKit [73], among others. Python-igraph was chosen as the main package for testing due to its speed, as it is based on C-code with a Python wrapper. It has a large community and is continuously updated. Additionally, it has a large library of implemented community detection algorithms, including most of the aforementioned algorithms and more. The Simula Ex3 [74] supercomputer was used for processing large graphs in terms of both quantity and size.

Based on different factors, such as running time and general performance of the algorithms tested during the exploration phase, four were chosen to be used in further iterations; SCoDA, Infomap, Leiden, and Louvain. These are described in detail in Chapter 3.

Chapter 12

Comparing Off-Line Algorithms

The off-line algorithms Infomap, Louvain and Leiden, were chosen based on the findings during the exploration phase. In contrast, SCoDA was chosen as the most promising on-line algorithm, due to its speed, detection of clusters and possibilities to modify and extend. The following chapter will look at how the three off-line algorithms differ from each other and what that means in a practical sense. While Leiden and Louvain have many similarities, with Leiden being an extension of Louvain, and InfoMap being entirely different, it is important to look at the characteristics of these algorithms in order to be able to compare them. Table 12.1 shows the characteristics of the three off-line algorithms.

Characteristic	**Infomap**	**Louvain**	**Leiden**
Concept	Random walk	Modularity Optimization	Modularity Optimization
Complexity	$O(k^2 * E)$ [75]	$O(m)$ [1]	$O(n \log k)$ [2]
Overlapping	Yes	No	No
Directed	Yes	No	Yes
Weighted	Yes	Yes	Yes

Table 12.1: *Off-Line Clustering Algorithms Overview*

The Louvain and Leiden algorithms are centred around modularity optimization, while Infomap is based on random walks. While all three algorithms perform well and can be used on large networks, Louvain and

[1] Not available, but assumed to be $O(m)$, where m is the number of edges [44]

[2] Not available, but reported to be 2-20 times faster than Louvain [40] and is assumed to be $O(n \log k)$, where n are the nodes in the graph and k is the average degree [44]

Leiden, due to the use of modularity as opposed to Infomap, cannot be used on overlapping networks. The asymptotic running time was not available for neither Louvain or Leiden algorithms at the time of writing. Vincent Traag, however, [44] cites that the running time of Louvain is assumed to be $O(m)$, where m is the number of edges. Due to several improvements made to the Leiden algorithm, it is reported to be 2 - 20 times faster than Louvain [40] and is be assumed to be $O(n \log k)$, where n is the number of nodes and k is the average degree [44].

	Infomap	**Louvain**	**Leiden**
Time (undirected)	4.42 min	2.27 sec	1.49 sec
Time (directed)	18.5 min	N/A[1]	5.78 sec[2]
Communities detected	2870	79	77
Modularity (undirected)	0.44	0.80	0.81

Table 12.2: *Performance of the algorithms*

Figure 12.2 shows the performance of the three algorithms. The tests were performed on a personal computer[3]. Each algorithm was run on a Twitter graph from Stanford Large Dataset Collection (SNAP) [56]. In order to be able to compare the algorithms, each graph was converted to an undirected graph. This was done due to the fact that modularity in its base form cannot be calculated on directed graphs. There are several papers [76, 77] suggesting modifications to the modularity calculations for calculating modularity on directed graphs, but this is deemed as being outside the scope of this thesis. This also resulted in a faster runtime for Infomap, whose runtime went from 18.5 minutes directed to 4.42 minutes undirected, corresponding to a 76 % reduction.

There are currently two implementations of the Leiden algorithm available through the Python-igraph library, built-into the library or as an extension created by the authors of the Leiden algorithm paper [40]. While the one built-into the Python-igraph (verions 0.8.x) is generally faster, the extension has the advantages of working on directed and weighted graphs. These will be used interchangeably, based on the use case, with the built-in version being preferred primarily, due to its speed.

The Leiden algorithm had the best performance, both in terms of modularity score and runtime. Additionally, the modularity score was about half of the Leiden score. Leiden and Louvain methods appear to perform equally well, with Leiden finding slightly fewer communities while getting practically the same modularity score.

The poor modularity score of Infomap was expected as it focuses on detecting flow in the network, rather than maximizing the modularity score. The prohibitively large runtime, however, makes it practically unsuitable for on-line scenarios where speed is essential.

[1]The Leiden extension of Python-igraph can be run on directed graphs and the score is based on that.

[3]ThinkPad X1 Carbon Gen6, Intel i7-8550U, 16 GB RAM

Chapter 13

Off-Line Algorithms on Incremental Graphs

13.1 Implementation

The tests performed in Chapter 12 show how the algorithms perform in their *natural scenario*, i.e. off-line. In order to check how the three algorithms perform when a graph is streamed, special tests must be run. The test environment should be similar to the live environment in which the algorithms will be used. However, when working in an unknown field, it can be beneficial to start with a more simplified version of the problem. In order to achieve this, the Stochastic Block Model-graph (SBM), mentioned in Chapter 10.5, was used for this test. The graph was generated in such a way that the communities are clearly defined. This means that nodes inside of clusters had many connections amongst themselves and few connections between nodes in different communities, which should be quite easy for algorithms to cluster.

Both modularity-based algorithms have a respectable performance when run once, but during a streaming scenario, the algorithm could potentially be run an infinite amount of times. Running them each time edges are added is not a viable solution, as the runtime would make the fastest algorithms use an unreasonable amount of time. This notion demands solutions where the algorithms are not run every time but in intervals. To prove that reclustering on every event is inefficient, a simple test was created where each time a new edge of the SBM network was received, the chosen clustering algorithm would be rerun. The results were recorded and are presented in Table 13.1 as *Base time*. The tests with improved performance, presented as *Time*, were done by going through an edge list edgewise and adding one node pair (i, j) at a time to a graph. Each added edge was used to find a sub-graph of the immediate neighbours of these nodes. The modularity score of the graph was continuously monitored, and when below a threshold, the chosen community detection algorithm was re-run on the graph. Otherwise, the new nodes were placed into communities that they most likely to be in. This community was calculated by looking at the communities of their neighbours and finding

the most common in the nodes' neighbourhood. The new nodes were then assigned to that community. This approach cut down the runtime of the base-case considerably (decrease of 55 - 74 %). All three algorithms were run with the same parameters to guarantee a fair comparison. The modularity score, the number of clusters discovered and runtime were recorded.

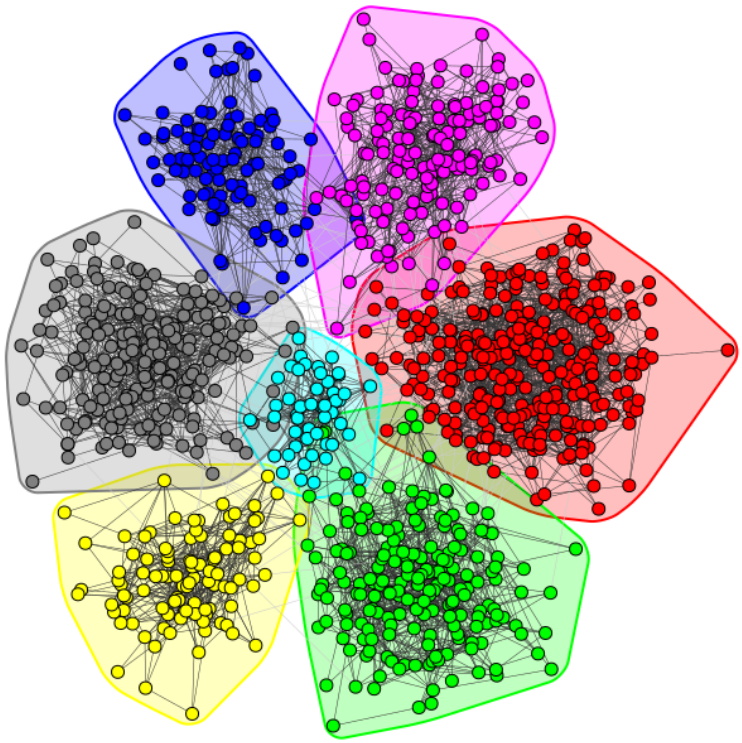


Figure 13.1: *Communities discovered by the Leiden algorithm*

13.2 Evaluation

Table 13.1 shows the performance of the algorithms when testing on a personal computer[1]. All three algorithms achieved the same modularity score of 0.79, and all three detected the same number of communities. Additionally, all three algorithms found the correct number of nodes in each community, except Infomap, which only misplaced one node. When comparing running times, Leiden and Louvain algorithms were the most efficient, taking just a fraction of the time it took to run Infomap; 3 and 5 seconds respectively, versus 19 minutes runtime of Infomap. Figure 13.1 depicts the clustering result of the Leiden algorithm. Due to the clustering results being nearly identical, the figure represents the results of the other algorithms as well.

A fact worth mentioning is that the modularity score of the streaming-like scenario was the same as running them in the normal fashion, i.e.

[1]ThinkPad X1 Carbon Gen6, Intel i7-8550U, 16 GB RAM

running it once on the whole graph. This appears to point to the success of the stream-based reclustering method. The approach of cutting down runtime by "guessing" the incoming node's community made it unnecessary to recluster the graph after each event, resulting in faster runtimes.

Algorithm	Base time	Time	Clusters	Modularity
Leiden	10 s	3 s	7	0.793
Louvain	19 s	5 s	7	0.793
Infomap	45 m 34 s	19 m 38 s	7	0.793

Table 13.1: *Performance of the algorithms on stream-based graph*

13.3 Summary

The performance of Infomap in terms of runtime in the simulated streaming scenario was disappointing. With all three algorithms running with the same parameters, the Infomap was not only generally slower but also had to be run more often than Louvain and Leiden to keep the modularity score above the threshold. While the latter was expected, the runtime of Infomap compared to Leiden was 392 times longer. Despite the possibilities Infomap provides in terms of understanding a network better, by looking at its flow, the prohibitively large runtimes make it impossible to use on huge graphs.

Chapter 14

Comparing Off- and On-Line Algorithms

14.1 Implementation

Until this point, this thesis has primarily focused on the testing of off-line algorithms in various scenarios. The topic of this chapter is to compare the general performance of off-line and on-line algorithms. The **SCoDA** algorithm was chosen as the on-line algorithm, as the paper [45] about it, presented in Chapter 9, suggested it to be extremely fast and able to find good clusters. The C++ implementation of the algorithm [78], developed by the creators of the algorithm, Alexandre Hollocou et al., was used. **Leiden** was chosen as the off-line algorithm, due to its performance during the tests, described in Chapters 12 and 13. The Leiden implementation in the Python-igraph library was used for comparison, also being a C++-implementation.

Looking at the description of SCoDA there are two obvious problems with the algorithm - the *required shuffling of the edge list* and the *importance of choosing the correct degree threshold*. Hollocou et al. describe in the paper that in order to get the best results from SCoDA, the edges of the network must be shuffled. Furthermore, choosing the correct degree threshold is critical for the discovery of good clusters. In a real-life streaming scenario edges would be arriving randomly, with a higher probability of "popular" nodes receiving more edge and therefore arriving "earlier", as per definition presented in the paper. The requirement of shuffling edges is therefore not as pertinent to the real-life scenario. The degree threshold, however, may pose a challenge when the network characteristics are not known beforehand.

Because the SCoDA-implementation only had a manual degree threshold input, the algorithm was run 50 times for each graph, and the degree threshold number that resulted in the highest modularity score was recorded.

Both algorithms were run on a set of 5 895 graphs. The set was a collection of different graphs collected by the supervisor of this thesis, Johannes Langguth, and consisted of DIMACS10, SNAP and other real-world and

synthetic graphs with highly varying characteristics. The modularity score for both algorithms was calculated was recorded. Additionally, the number of communities discovered was saved for comparison.

14.2 Results

	With Null-Values	**Without Null-Vales**
Number of graphs	5 895	1939
Average number of nodes	157 623	151 477
SCoDA average threshold	7.59	23.1
Leiden average modularity	0.20	0.62
SCoDA average modularity	0.05	0.15
Leiden average communities	32 343	75 977
SCoDA average communities	2 469	7 507

Table 14.1: *SCoDA and Leiden average values*

Due to the varying characteristics of the graphs tested, several networks were not suitable for the clustering algorithms to perform. The graphs that are unsuitable for clustering are, for example, densely connected graphs, meaning that each node is connected to every other node in the network. Clustering these graphs would be unreasonable, as the whole graph would be in one giant cluster. The special characteristics of these networks resulted in modularity score of 0 for both Leiden and SCoDA. These results will hereby be called *the null-cases* or *cases with null-values*. Table 14.1 presents the average values for the two cases - with and without modularity null-values. While it would be interesting to delve deeper into the null-value graphs in order to find out exactly what happened, these results are not useful when comparing the two algorithms and the results from these graphs have been omitted during further processing of data. "With Null-Values"-column shows average scores, including graphs where either of the algorithms resulted in modularity score of 0. These graphs are omitted in the "Without Null-Values"-column.

14.3 Evaluation

From Table 14.1 one can immediately see that the Leiden algorithm performs much better, achieving a higher average modularity score of 0.62 compared to 0.15 of SCoDA in the Non-null-value set. This also reflects on the number of times Leiden performed better, which is 1 766. SCoDA, on the other hand, performed better in just 143 cases. The two algorithms got the same score in 29 cases. In the cases where SCoDA performed better, its average modularity was 0.48, while Leiden had a modularity score of

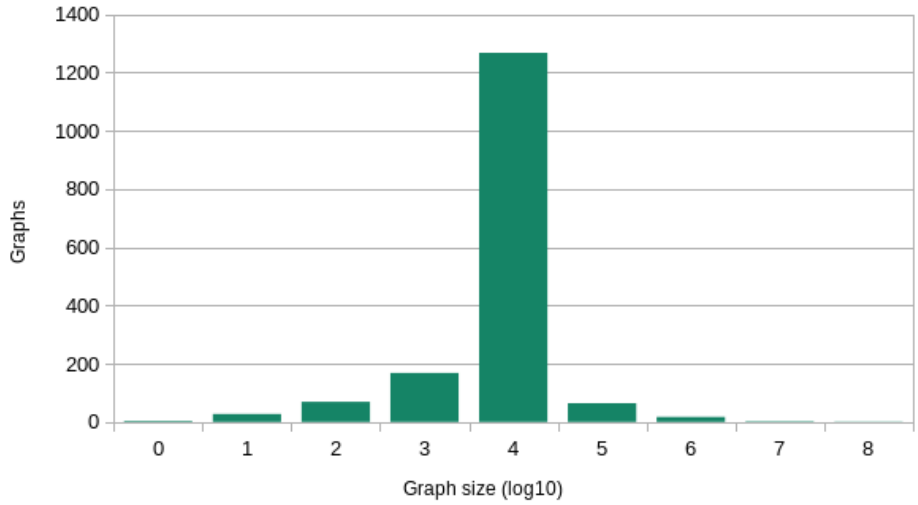


Figure 14.1: *Distribution of graphs by size*

0.11. In the opposite case - where Leiden performed better, the modularity scores were 0.65 for Leiden and 0.11 for SCoDA. In the rare case of the two being equal, the modularity was often high, with an average score of about 0.85. Furthermore, in 319 cases SCoDA failed to find any communities, meaning that its modularity score was 0, while Leiden's was positive. The aggregated average modularity of Leiden in those cases was 0.59. This points to the fact that many of the non-detected communities were defined relatively well. Further testing may be needed to see exactly why this happened, but this is considered out of scope for this thesis.

Figures 14.1, 14.2 and 14.3 present the findings graphically. The graphs in the figures are split into log10-bins for clarity, as sizes of the graphs range between 3 and 65 million nodes. Figure 14.1 shows the number of graphs in each bin, with the majority being in the $\log 4$-bin, meaning that the number of nodes in these graphs is around 10^4. Figures 14.2 and 14.3 show the performance of the algorithms on the test set. Figure 14.2 shows the average modularity score of Leiden and SCoDA respectively, partitioned by graph sizes. In order to depict the difference in performance more clearly, Figure 14.3 shows the share of each algorithms modularity score of their total sum in each bin, meaning that the more the bar is filled with the algorithm's colour, the better it performed on the graphs of that size.

While SCoDA generally had a worse performance than Leiden, Figure 14.3 shows that SCoDA is relatively competitive with the Leiden in many cases. SCoDA had similar results on graphs up to a certain size ($\log 6$). However, both algorithms had a dramatic drop in modularity on graphs in the $\log 5$-bin. It is not clear why this happened without analyzing individual graphs in the set. The fact that both algorithms had problems with this size points to the fact that the graphs in this bin may have had a topology unfit for clustering. After the drop, Leiden seemed to recover

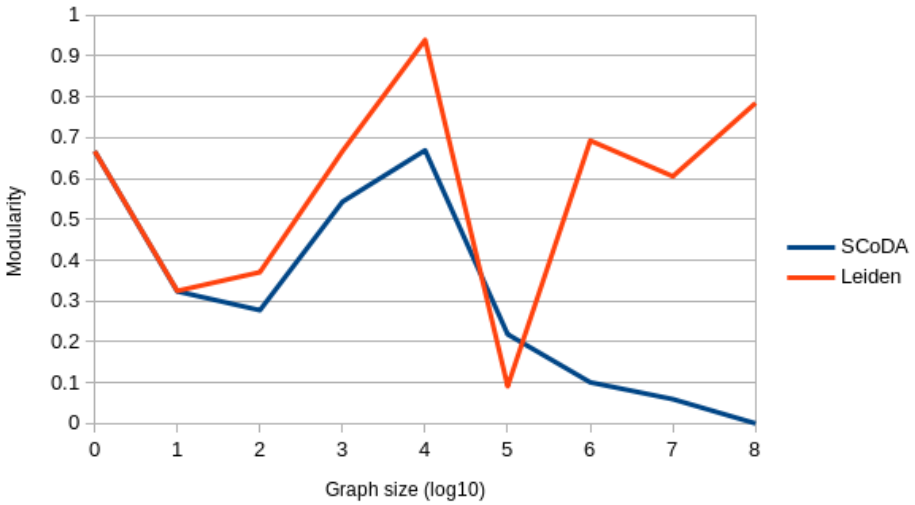


Figure 14.2: *Modularity scores of Leiden and SCoDA algorithms based on the sizes of graphs*

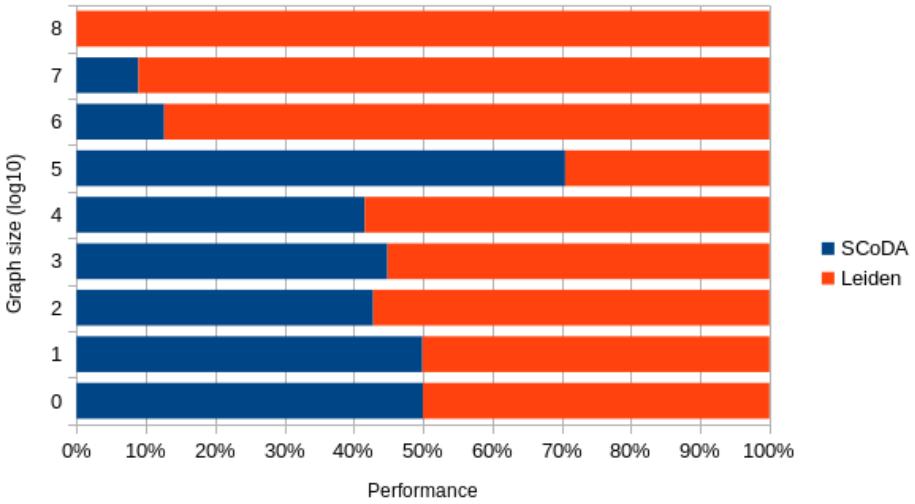


Figure 14.3: *Modularity scores of Leiden and SCoDA algorithms based on the sizes of graphs, in percent of the sum of each bin*

and perform well, while SCoDA's performance continued to degrade. This may point to the fact that SCoDA could not handle the sizes of these graphs and once it started making errors in clustering, it could not recover. It is, however, important to mention that SCoDA performed better than Leiden on the graphs in the log 5-bin, again pointing to the possible fact that the graphs found in that bin were hard to cluster using modularity optimization techniques.

It is worth mentioning that the optimal threshold value for SCoDA found in the tests differs from the one mentioned in the paper [45], which states that the best result is usually obtained at degree threshold values

between 2 and 4. For non-null results of SCoDA, the three values of the threshold that yielded best results in modularity values above 0 were 39, 48 and 49, with 23.1 being the average chosen threshold for all SCoDA runs. A possible reason for this may be that the tests conducted by Hollocou et al. primarily covered real-life networks from the SNAP-database, which have greatly varying characteristics compared to graphs that were tested in this chapter.

14.4 Summary

When looking at average statistics of the results of Leiden and SCoDA, the formar seems to be superior to the latter. After further investigation, however, SCoDA seems to provide acceptable results for graphs up to a certain size and, in some cases, is even able to outperform Leiden, which is a quite more complex and sophisticated algorithm. One reason for SCoDA being able to outperform Leiden may be the fact that some of the graphs used in testing had special structures, like lattices, where most of the nodes are connected with at least one edge. This may have "confused" Leiden and given an edge to SCoDA, which does not take into account the structure of graphs in the same manner. These types of networks are seldom seen in real-world clustering applications, and further tests will exclude these types of graphs, as the purpose of this thesis is to find graphs that can perform well in real-world scenarios. The findings also point to the fact that it is difficult to determine the right degree threshold without knowing the graph beforehand.

Based on the findings in this chapter, the main takeaway is that when it comes to graph clustering there is no silver bullet. While an algorithm may perform well on one type of graph, it may have poor performance on another. It is therefore important to gather as much information as possible and conduct thorough testing before committing to an algorithm. This is especially true when working with algorithms in situations they were not designed for. Going forward, the main goal will be to design an algorithm that works on on-line incremental graphs. The design decisions must reflect that.

Chapter 15

Designing an On-Line Incremental Algorithm

15.1 SCoDA-Leiden Algorithm

Based on the findings in the previous chapter and the discovered strengths and weaknesses in both Leiden and SCoDA, the initial thought was to merge the two algorithms in a way that combined the strengths of the two algorithms, while diminishing their weaknesses, leading to an algorithm that finds high-quality clusters in an on-line scenario. For this purpose, the Leiden and SCoDA algorithms were merged to create the **SCoDA-Leiden algorithm**.

The modifications were as follows: a given graph would be split into two partitions. The Leiden algorithm would cluster the first partition, and the result would be saved in the memory. This was done to simulate a case where some data about the graph is available. The second partition would be streamed edge-wise, and SCoDA would be run. While this happened, modularity score would be calculated every 10^i edges, where i is the length of the number of edges in the network, e.g. for edge count between 7 000 and 30 000, the modularity would be recalculated at 7 000, 8 000, 9 000, 10 000, 20 000, and 30 000 edges. If the modularity score dropped by 5 % or more, Leiden would be re-run on the whole graph and clusters would be updated. SCoDA would then continue running on updated communities. This approach was based on the observation that when running SCoDA, the modularity decline was relatively small when staring with good clusters. The steps of the algorithm are as follows:

1. Split an edge list p into two parts: $p1$ and $p2$, such that the all of the edges of p are either in $p1$ or $p2$.
2. Cluster $p1$ with Leiden and store the results in memory.
3. Add $p2$ to $p1$ edge-wise. For every 10^i edges, check if modularity has decreased by more than 5 %.
    - If yes: recluster the whole graph with Leiden.
    - If no: continue.

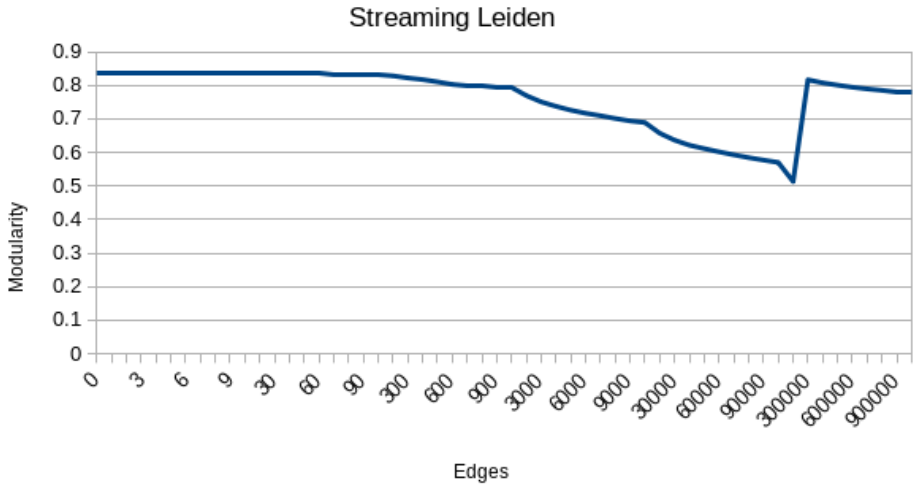


Figure 15.1: *Modularity of the SNAP Twitter graph as the number as edges from $p2$ are added to pre-clustered $p1$*

Figure 15.1 shows the modularity score of the clustering of the SNAP Twitter-graph as the $p2$ is added to $p1$ edge-wise. The split is done unequally, with $p1 = 30$ % and $p2 = 70$ % of the edge list. The increased modularity at 300 000 edges is due to the reclustering by Leiden. The drop just before the increase was more than 5 %, meaning that Leiden had to recluster the whole graph available at that time. The SCoDA then took over again, as can be seen by small, continuous drops in modularity from 400 000 edges until the end.

The SCoDA-Leiden algorithm had good modularity preservation, with the average modularity score being between 90 - 95 % in most cases. The runtimes, however, warrant a closer look. Due to the linear runtime of SCoDA, which the algorithm primarily uses, the asymptotic runtime for the best-case scenario is $O(n)$ plus the runtime of the first Leiden clustering (L). However, as seen in the tests conducted on SCoDA, as the graph gets bigger, the quality of clusters degrades, and Leiden would have to be re-run, adding the runtime of Leiden, each time this happens. The worst case of this algorithm, may, therefore, be prohibitively large. As the graph grows, so will the number of times Leiden has to be re-run (y for $y \in O(n)$). This results in the asymptotic runtime of $O(yL)$, which is prohibitive to be used efficiently on large graphs. In order to be able to construct a good on-line incremental algorithm, we must limit the worst case running time.

15.2 Requirements for clustering incremental graphs

Conducting clustering on incremental graphs allows us to set some requirements on the algorithms being run. When deciding which requirements should be chosen, we conduct a thought experiment about what challenges we will be faced with.

Let graph G be of several trillions nodes. When choosing to cluster it, the size of the graph may be prohibitive for running a clustering algorithm at once, regardless of how speed- and memory-efficient it is. The graph must therefore be split into parts that fit into memory and may be clustered within a reasonable time frame. The result of this is that the graph will become incrementally bigger by adding new partitions to it. These partitions must be clustered and merged in a way that leads to minimum loss of goodness. While clustering the individual partitions is as trivial as running a chosen clustering algorithm on it, the merging of the clusters in the memory with the clusters of incoming partitions provides several challenges, like how to assign clusters of the new partition to the ones in the memory and when to update clusters of the nodes that have already been assigned to a cluster. This notion leads us to the conclusion that the merging algorithm plays a crucial role in the process of clustering incremental graphs.

Following requirements were proposed as a result of this thought experiment:

1. The original graph must be stored in an efficient manner
2. The incoming graphs must not be larger than the available memory
3. The incoming graph must be added to the original graph
4. The time to add the new graph must be polynomial in the size of that graph
5. The success of this addition can be measured in the loss of modularity, as compared to clustering the graph as a whole.

The aforementioned requirements are set up as guidelines that will be used in order to build algorithms to be used on on-line incremental graphs.

Chapter 16

Designing a Merging Clustering Algorithm

16.1 Introduction

16.1.1 Process

The proposition of requirements for creating a good on-line incremental clustering algorithm, presented in Chapter 15, demonstrated the need to develop and test different approaches in an efficient manner. In order to achieve this, a framework that would allow to test and compare different merging and clustering algorithms was developed. This development was split into three steps:

The first step was to create network merging strategies and test them out on a two-way split of a given graph. In order to be able to compare the different approaches, the modularity of the merged result would be compared to the modularity provided by running Leiden on the whole graph. The performance of the algorithms was rated based on the minimum loss of modularity after completing the graph merging, compared to Leiden's score.

The second step was to extend the partitioning of the graph into several splits, chosen by the user, and perform the merge of these partitions in a way that the merged graph would be the exact replica of the original graph. The clustering result of the merge served as the base for comparing the merging algorithms in order to find the best approaches.

The third step was to choose the algorithm(s) with the best performance and, if possible, explore ways to make them better and more efficient.

16.1.2 Definition of terms

In order to describe the tests in a precise manner the following terms will be used for the rest of the thesis:

- G: $V * E$, where V are the vertices of the graph and E are the edges connecting the vertices. G is attained through the input edge list. The baseline modularity will be calculated by running the Leiden algorithm on G.

- P_k: let G be split into k partitions. P_k is the k-th partition of G. This graph consists of vertices, V_{Pk} and edges E_{Pk}.

- $\overline{G}_k$: obtained by merging partitions P_{k-1} and P_k of G: $\overline{G_k} = \overline{G}_{k-1} \cup P_k$. The result of merging of P_{k-1} and P_k is either $\overline{G_k}$ or G. $\overline{G}_k$ consists of vertices $\overline{V}_k$ and edges $\overline{E}_k$.

16.2 2-split merge

The first step of creating the testing framework was to achieve a successful merging and clustering of a graph split of two chunks. The edge list of a chosen graph G was split into two parts P_1 and P_2, where the user could change the partition split size between 0 and 100 % in terms of the number of edges, meaning that if P_1 was chosen to be 30 %, the size of P_2 would be the remaining 70 %. P_1 and P_2 were then saved as two separate edge lists. P_1 was clustered by a preferred algorithm (Leiden in this case), resulting in $\overline{G}_1$, and the clusters were stored in memory[1], together with its edge list. Additionally, some data about the network topology, like the degree of each node, was stored for testing purposes.

P_2 was also read and clustered by a chosen algorithm, resulting in $\overline{G}_2$. $\overline{G}_1$ and $\overline{G}_2$ were then merged using different techniques. The modularity of the approach and the number of resulting communities was stored, in addition to the data from a Leiden run on G. Furthermore, the runtimes were also recorded for testing purposes. The framework allowed for multiple runs and returned average scores if multiple runs were conducted. The average scores represented different partition sizes varying from *no-input graphs*, i.e. $P_1 = G \;\;\&\;\; P_2 = \emptyset$, to *no-starting graphs*, i.e. $P_1 = \emptyset \;\;\&\;\; P_2 = G$. The size of the split was determined automatically by the preferred number of runs. A test of five runs, for example, would split G into five sets of P_1 and $P2$ of different sizes, denoted in percent of edges of G, where `Set 1` would be $P_1 = G \;\;\&\;\; P_2 = \emptyset$, `Set 2` would be $P_1 = 0.2G \;\;\&\;\; P_2 = 0.8G$, `Set 3` would be $P_1 = 0.4G \;\;\&\;\; P_2 = 0.6G$, and so on. As the runtime of the merging algorithm is dependent on the input graph, as described in point 4 of requirements in Chapter 15, the average runtime would be a good measure for comparison between different merging strategies. In order to guarantee that the data is not known beforehand, the edge lists were shuffled each time before being split.

[1] Each node was represented as a key-value pair in a dictionary

16.3 Merging approaches

16.3.1 Phases of merging

The main limiting factor of the potential merging algorithms was that they did not have information about the whole graph G, as per the aforementioned requirements in Chapter 15. An algorithm only had access to limited part of the $\overline{G}_{k-1}$, in addition to the P_k. The incremental merging algorithm was split into the two following phases:

Phase 1: Pre-clustering and Merging Phase 1 is centered around the notion of clustering P_k and roughly merging it with $\overline{G}_{k-1}$ into the resulting $\overline{G}_k$ without loss of information. This phase consists of reading P_k and running a preliminary clustering on it. The processing time of Phase 1 depends almost exclusively on the size of P_k.

During Phase 1, an algorithm would be run in order to cluster and merge P_k with $\overline{G}_{k-1}$. The merge was performed by comparing the number of nodes in $\overline{G}_{k-1}$ with the number of nodes in P_k. If $\overline{G}_{k-1}$ was larger than the P_k, only the nodes that were not present in $\overline{G}_{k-1}$ were added to the clusters C_k of $\overline{G}_k$. If P_k was larger, the nodes that were present in the original clustering $\overline{C}_{k-1}$ of $\overline{G}_{k-1}$ were moved to the communities discovered by clustering of P_k. **Algorithm 16.1** shows the pseudo-code for this approach.

Algorithm 16.1 Base merging algorithm

```
Ḡ_k ← Ḡ_{k−1} ∪ P_k
node_community = dict{node : community}
C_k ← leiden(P_k)
if Ḡ_{k−1}.size > P_k.size then
    for node in C_k do
        if node_community[node] is null then
            node_community[node] ← C_k.community
        end if
    end for
else
    for node in C_k do
        node_community[node] ← C_k.community
    end for
end if
```

Phase 2: Refinement Once Phase 1 was completed, the refinement phase would start. The purpose of this phase was to refine the clustering produced in Phase 1 in order to achieve the minimum loss of modularity. In this phase, the algorithm had some limited access to $\overline{G}_k$, like the neighbourhood of nodes of the incoming graph.

16.3.2 Refinement approaches

Ten different merging approaches were tested; the base merging algorithm from Phase 1 and nine refinement algorithms for Phase 2 were developed. It is important to mention that the algorithm for Phase 1 is a stand-alone algorithm, which produces a clustering. However, due to its knowledge of the graph G being strictly restricted to the incoming partitions P_k, the results were sub-optimal. It was, however, deemed to be a good starting place for the refinement phase and was therefore used as a base case. A refinement algorithm that would degrade the modularity score of the base algorithm was considered to be non-functional, as it would only add to the runtime without any positive effect on the clustering performance.

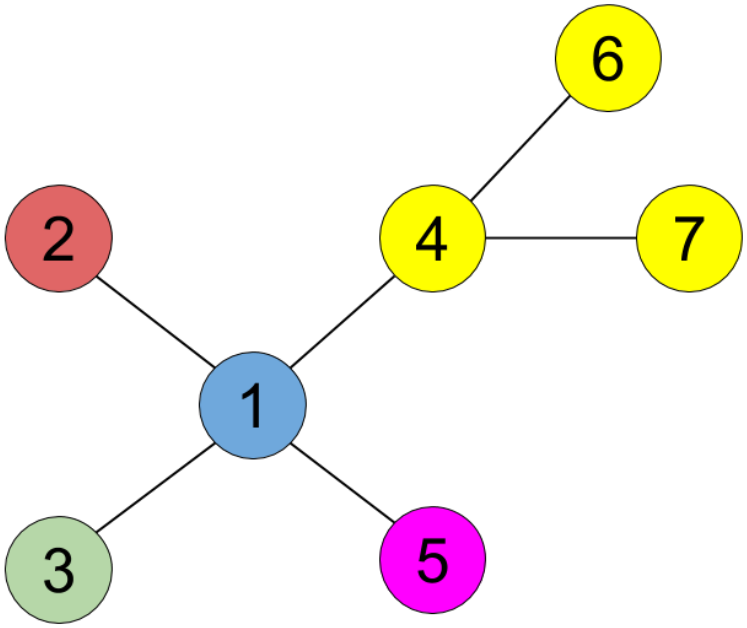


Figure 16.1: *The graph used as an example of refinement-phase strategies*

The following algorithms were tested for Phase 2. Figure 16.1 is used to describe how the refining algorithms work. Each algorithm is superseded by an example. In all examples P_k consists of nodes 1 - 7, which are named by their processing order and coloured by their cluster. Nodes 2 - 5 are neighbours of node 1.

0. No algorithm

The result from Phase 1 is used directly, with no additional merging strategy was used. Phase 2 is skipped.

1. Node-wise merge

For each node-pair (u, v) connected by an edge in P_k, the one with the lower degree joins the the one with the higher degree. This approach tries to

combines the clusters of nodes with low degree with the ones with high degree.

Algorithm 16.2 1. Node-wise merge

```
for node in P_k do
    neighbours ← get_neighbours(node)
    for neighbour in neighbours do
        if node_degree[node] > node_degree[neighbour] then
            node_community[neighbour] ← node_community[node]
        else
            node_community[node] ← node_community[neighbour]
        end if
    end for
end for
```

Example: nodes 2, 3, 4 and 5 would change to blue cluster. Following this nodes 6 and 7 would change their cluster to blue, as node 4 has been changed.

2. Node-wise merge with factor

For each node-pair (u, v) in P_k if degree of u is at least two times higher than the degree of v, then v joins the community of u. In order to restrict the merging of the communities that should not have been merged, a factor of two is introduced. A node will only change communities if at least one of its neighbours is at least two times more "important" than itself. The factor may be changed, but the factor of 2 resulted in overall best performance in initial tests.

Algorithm 16.3 2. Node-wise merge with factor

```
for node in P_k do
    neighbours ← get_neighbours(node)
    for neighbour in neighbours do
        if node_degree[node] > 2 * node_degree[neighbour] then
            node_community[neighbour] ← node_community[node]
        end if
    end for
end for
```

Example: nodes 2, 3 and 5 would change to blue community as node 1 has at least 2 times as high degree as these nodes. Node 4, 6 and 7 would not change their communities, as node 1 has only 4/3 the degree count.

3. Ballot with one vote per node

Every neighbour v of node u in P_k votes on which community the neighbourhood should be in, and votes for its own community. The

neighbourhood changes to the community with most votes. In case of ties, community is chosen at random.

Algorithm 16.4 3. Ballot with one vote per node

```
for node in P_k do
    neighbours ← get_neighbours(node)
    ballot ← dict{}
    for neighbour in neighbours do
        ballot[node_community[neighbour]] ← +1
    end for
    chosen_community ← max(ballot)
    node_community[node] ← chosen_community
    for neighbour in neighbours do
        node_community[neighbour] ← chosen_community
    end for
end for
```

Example: the community of nodes 1, 2, 3, and 5 would be chosen at random, as each community received one vote. Node 4, 6, and 7 would stay in the yellow community.

4. Ballot with one vote per degree

Same as Algorithm 3, but every neighbour v gets votes proportional to its degree. The group changes to the community with most votes. In case of ties, random community is chosen.

Algorithm 16.5 4. Ballot with one vote per degree

```
for node in P_k do
    neighbours ← get_neighbours(node)
    ballot ← dict{}
    for neighbour in neighbours do
        ballot[node_community[neighbour]] ← +node_degree[neighbour]
    end for
    chosen_community ← max(ballot)
    node_community[node] ← chosen_community
    for neighbour in neighbours do
        node_community[neighbour] ← chosen_community
    end for
end for
```

Example: nodes 2, 3, 4, 5, 6, and 7 would change their community to blue, as node 1 has four votes, due to its degree.

5. Neighbourhood-to-community link counting

For each neighbour v of a node u, count the number of times each community represented. Move u to the community represented the most.

In case of ties, the community is chosen at random. This idea is similar to Algorithm 3, but in this case only one node changes communities instead of the whole neighbourhood.

Algorithm 16.6 5. Neighbourhood-to-community link counting

```
for node in P_k do
    neighbours ← get_neighbours(node)
    neighbour_links ← dict{}
    for neighbour in neighbours do
        neighbour_links ← +1
    end for
    best_community ← max(neighbour_links)
    node_community[node] ← best_community
end for
```

Example: node 1 would change the community to either red, green, yellow or pink. Node 2 would change its community based on what was chosen by node 2. The same is the case for node 3 and 5. Nodes 4, 6, and 7 would stay in yellow community.

6. Edge-wise merge by degree

For every edge in P_k, the node with the lower degree joins the node with the higher. This approach is inspired by the SCoDA algorithm, where the edge decides whether one node should join the community of another.

Algorithm 16.7 6. Edge-wise merge by degree

```
for edge in P_k.edgelist do
    u, v ← edge
    if node_degree[u] < node_degree[v] then
        node_community[u] ← node_community[v]
    end if
    if node_degree[v] < node_degree[u] then
        node_community[v] ← node_community[u]
    end if
end for
```

Example: nodes 2, 3, 4, and 5 would join the blue community, as node 1 has the highest degree. Nodes 6 and 7 would also change to the blue community, as node 4 would be blue by the time their edges are processed.

7. Community-wise merge by degree

For every community discovered in P_i, the one with the lowest sum of degrees joins the one with the highest. This algorithm explores the relationship between intra- and interlinks and is also the most technical

and resource heavy. A threshold variable may be set in order to stop communities from merging too aggressively.

Algorithm 16.8 7. Community-wise merge by degree

```
inter_links ← get_interlinks(P_k.edgelist)
intra_links ← get_intralinks(P_k.communities)
for community_link in interlinks do
    if intra_to_inter_link_ratio > threshold then
        if community_links.c1 < community_links.c2 then
            node_community[c1.nodes] ← c2
        end if
        if community_links.c2 < community_links.c1 then
            node_community[c2.nodes] ← c1
        end if
    end if
end for
```

Example: nodes 4 - 6 are considered a cluster with the intra-cluster edge count of 2. Node 1 would therefore join the yellow community, as the sum of the blue intra-cluster edges is 0. Nodes 2, 3 and 5 would also shift to yellow community as the result of the first shift, with the yellow community continuously gaining more intra-cluster edges.

8. Leaf aggregation

For every edge between a node-pair (u, v), in P_k, if v has a degree of 1, it joins the community of u. This algorithm works on the assumption that the clustering of Phase 1 is adequate and only tries to "clean up" the leaf nodes.

Algorithm 16.9 8. Leaf aggregation

```
for edge in P_k.edgelist do
    u, v ← edge
    if node_degree[v] < 1 then
        node_community[v] ← node_community[u]
    else if node_degree[u] < 1 then
        node_community[u] ← node_community[v]
    end if
end for
```

Example: nodes 2, 3 and 5 are leaf nodes, and would join the blue community. Nodes 6 and 7 are already in yellow community and would stay there.

9. Eigenvector centrality

For every edge between a node-pair (u, v) in P_k, the node with the lower eigenvector centrality joins the one with the higher. The eigenvector

centrality is the measure of influende of a node on a network. It may therefore be a good way to ensure that the communities are based around the most "important" nodes.

Algorithm 16.10 9. Eigenvector centrality

```
ec ← get_eigenvector_centrality(P_k)
for edge in P_i.edgelist do
    u, v ← edge
    if ec[v] < ec[u] then
        node_community[v] ← node_community[u]
    else if ec[u] < ec[v] then
        node_community[u] ← node_community[v]
    end if
end for
```

Example: the eigenvector centrality of nodes in the graph would be: node 1: 1.0, node 2: 0.459, node 3: 0.459, node 5: 0.459, node 4: 0.796, node 6: 0.366, node 7: 0.366. Following this, nodes 2, 3, 4, and 5 would join the blue community. Nodes 6 and 7 would change to the blue community due to node 4 changing previously.

16.3.3 Testing 2-split of graphs

The different merging strategies were run on three different graphs a number of times and the results were recorded. The three graphs chosen were Zachary's Karate Club graph, The stochastic block model (SBM) graph, and SNAP Twitter graph. These graphs were chosen due to the variation of size and difficulty of clustering. Tables 16.1, 16.2 and 16.3 present these findings.

Note: When talking about the different algorithms, their corresponding numbers will be used, e.g. if the data about the *Base algorithm* was described, the algorithm would be presented as number 0. This is done to clearly differentiate between the different algorithms, due to some of them having names similar to each other.

The algorithms 5, 8 and 0 had the highest modularity preservation and the lowest running times. The algorithm 0, which does not use any additional merging techniques and was intended to be used as the base case for comparison, had the best "modularity preservation per second", i.e. the highest modularity divided by running time, of all algorithms. It is interesting to see that algorithm 8 (leaf aggregation) has the same score as algorithm 0, meaning that it does not seem to have any effect, which may indicate that there are few to no leaves with differing communities to the ones they are attached to. This may be due to the guarantees that Leiden gives about goodness of clusters [40].

Another interesting result, presented in Table 16.3, is the fact that all three algorithms actually performed better than the base Leiden in the

case of the SNAP Twitter graph. While the difference is negligible and may be attributed to advantageous shuffling, the fact that this happened consistently, with up to 10 re-runs per algorithm, is interesting. Further testing is required to discover whether more gain can be achieved.

The framework used for these tests is flexible and the clustering algorithms in both initial clustering and the clustering of the incoming graph may be switched out. It may be interesting to see whether two different algorithms may be combined in order to achieve even better scores. Another interesting test would be to see whether splitting the incoming graph into several smaller chunks would influence the merging process. These tests will be presented in the next chapters of this thesis.

Merging strategy	**Modularity**	**Communities**	**Runtime (10^{-3} s)**
Default Leiden	0.42	4	0.4197
5	0.38	4.17	0.9255
0	0.36	4.38	0.5194
8	0.36	4.35	0.7835
7	0.32	3.52	0.5856
2	0.28	4.2	0.8836
1	0.27	2.27	0.8285
6	0.26	3.96	0.6444
9	0.19	4.05	1.1990
3	0.05	1.86	0.9418
4	0.03	1.68	0.7801

Table 16.1: *Merging strategies on the Zachary's Karate Club-graph*

Merging strategy	**Modularity**	**Communities**	**Runtime (10^{-3}s)**
Default Leiden	0.79	7	9.91
0	0.79	8.43	12.60
8	0.79	8.59	12.95
5	0.79	8.0	19.39
2	0.78	7.34	19.03
6	0.75	7.71	13.98
1	0.7	7.01	18.61
9	0.63	8.2	21.87
3	0.61	7.14	21.80
7	0.56	6.24	18.55
4	0.54	6.97	22.36

Table 16.2: *Merging strategies on the SBM-graph*

Merging strategy	**Modularity**	**Communities**	**Runtime (s)**
Default Leiden	0.81	79	1.76
0	0.82	106.5	9.906
8	0.82	106.1	10.04
5	0.82	105.5	14.86
7	0.66	85.2	835.5
2	0.34	56.8	14.47
3	0.23	32.5	123.1
6	0.2	88.0	11.25
1	0.052	20.7	15.45
9	0.025	91.0	13.38
4	0.012	15.3	103.0

Table 16.3: *Merging strategies on the SNAP Twitter-graph*

16.4 Testing k-split of graphs

16.4.1 Approaches for base merging algorithm

In order to test how the merging algorithms work when G is split into several partitions, the testing framework was extended to split a given edge list into k-parts with roughly the same number of edges. The merging algorithm would then run on G and merge each partition P_k one by one. The modularity of the detected communities of the resulting $\overline{G}_k$ was recorded. The preliminary tests showed that the modularity score quickly dropped after the initial 2-split, described in previous section. The baseline merging algorithm, or *Base algorithm*, presented in Section 16.3.1, is the integral part of the merging process, as it serves as both the baseline for testing other algorithms and also as a base for the other algorithms to build upon. In order for the other algorithms to work, this phase must produce the best clusters possible. Several tests were run in order to establish the best possible way to conduct this. The different approaches were given names, presented below in bold, used in further testing and presentation of results. In order to understand the subtle differences between approaches, we introduce the notion of *Cluster IDs*, which are used in by Phase 1 to keep track of which nodes are in which communities. When no clusters have been discovered, the Cluster ID is 1. As new clusters are discovered the starting Cluster ID is updated, so in the following merge of P_{k+1}, the Cluster ID will start the the number of clusters of P_k + 1. The following approaches were tested:

Overwrite Intermediate cluster IDs of P_k, used for keeping track of clusters of $\overline{G}_k$, always start at 1, possibly overlapping with clusters of $\overline{G}_{k-1}$. Every node of P_k is placed in the communities found by clustering P_k, whether they have already been clustered or not. This results in nodes being moved between communities more often, as P_{k+1} will often change

the communities from previous merges.

Add Cluster IDs of P_k always start at 1, possibly overlapping with clusters of $\overline{G}_{k-1}$. Nodes that have already been clustered in $\overline{G}_{k-1}$ are not moved. A node from P_k is added to clustering of $\overline{G}_k$ only if it has not been seen previously. This will often results in less movement in Phase 1.

Update Cluster IDs of P_k start from the number of already discovered clusters. Nodes that are already present in the clustering of $\overline{G}_{k-1}$ are moved to their new clusters obtained through clustering of P_k. This approach may result in splitting of stable clusters if nodes in P_k are revisited often.

Append Clusters IDs of P_k start from the number of already discovered clusters. Nodes that are already present in the clustering of $\overline{G}_{k-1}$ remain in their clusters. Nodes from P_k that have not been seen before are added to new clusters. This approach may create many small clusters, as after some time, only a small subset of nodes in P_k will not have been seen previously.

Mix Described in Algorithm 16.1. If P_k is larger than the $\overline{G}_{k-1}$, nodes that are already present are moved to their new clusters, discovered by clustering P_k. Otherwise, only nodes not present in $\overline{G}_{k-1}$ are added to new clusters.

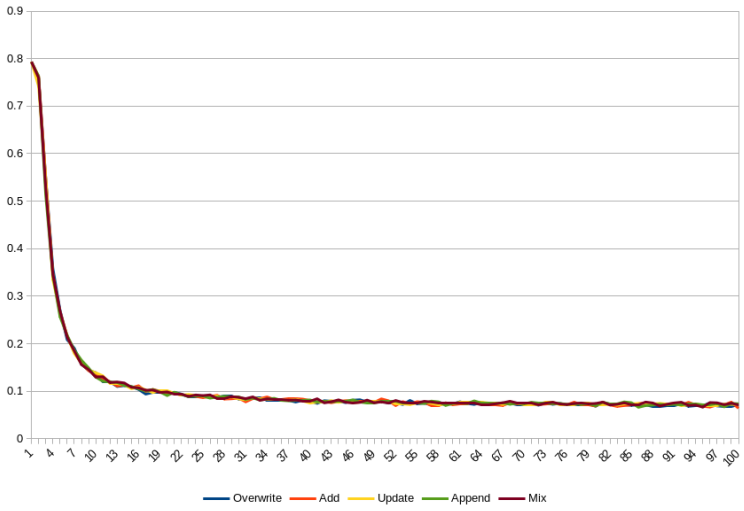


Figure 16.2: *Performance of base merging algorithm variations on k = 100-split of the SBM-graph*

Figure 16.2 depicts the modularity scores after merging a k-split, where the x-axis shows the k-th split and y-axis show the resulting final modularity. The SMB-graph, also used in Chapters 13, 14 and 15, was used for testing the different variations of the base merging algorithm with $k = 100$. It appears that there is little difference between the methods, with the mix method performing just slightly better than the others. It is also the

most versatile method, as it takes into account the sizes of P_k and $\overline{G}_{k-1}$. It was therefore chosen as the primary merging algorithm.

16.4.2 Conducting k-split test

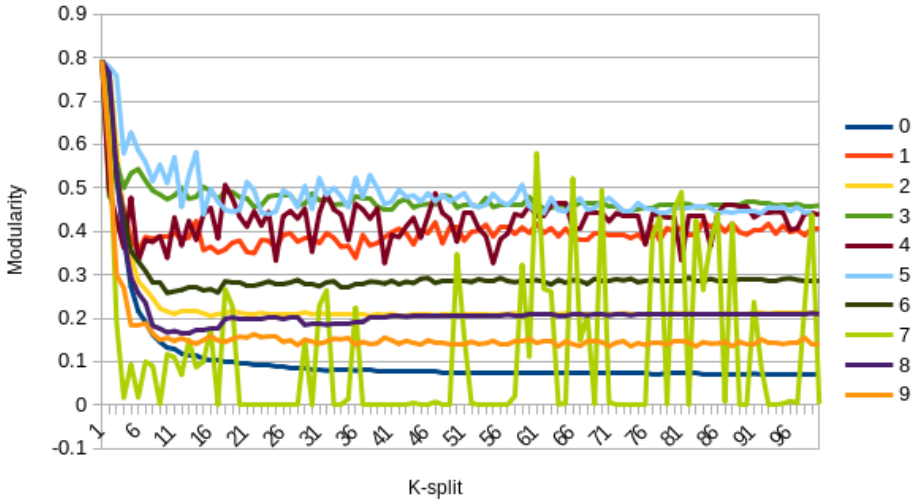


Figure 16.3: *Modularity of different merging algorithms of the SMB-graph as the value of k gets bigger*

Merging strategy	**Modularity**	**Communities**	**Runtime (s)**
Default Leiden	0.79	7	-
5	0.53	82.25	0.1127
3	0.47	16.56	0.2274
4	0.43	5.93	0.2164
1	0.4	28.2	0.2022
6	0.3	107.58	0.1662
2	0.23	265.15	0.2025
8	0.22	280.79	0.1668
9	0.16	332.61	0.2076
7	0.12	17.81	0.209
0	0.11	682.02	0.1721

Table 16.4: *Merging strategies on k-split of SBM-graph (k = 100)*

After choosing the approach for the *Base algorithm* (0), the next step was to test other algorithms with $k > 2$. For a preliminary test, the SBM-graph with $k = 100$ was chosen, meaning that the merge algorithm was run 100 times with k getting increasingly larger, while recording the runtime and results of each approach at each value of k. Figure 16.3 presents the average performance of the different algorithms as the value of k

increases. For each run, the edge list was split into k parts and the chosen merging algorithm was run iteratively, until k was reached. The modularity was then calculated and the running time, number of communities and modularity was recorded. The average of each of these entries was then calculated and the results were recorded. Table 16.4 presents the final result for the $k = 100$ on the SMB-graph.

In order to test the running time and performance on a larger real-world network, the k-split merge was run on the SNAP Twitter-graph with k-split of up to $k = 100$. Figure 16.4 and Table 16.5 present the results of these tests.

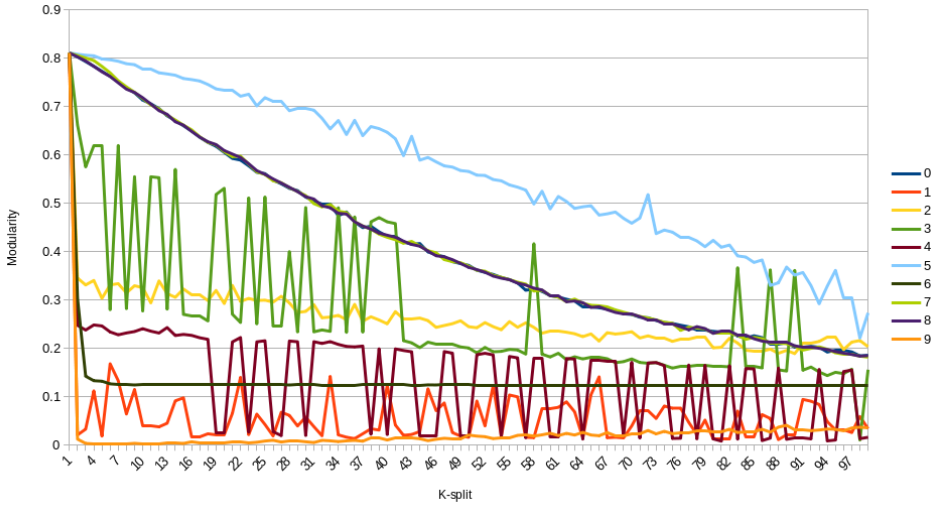


Figure 16.4: *Modularity variations of k = 100 on Twitter graph*

Merging strategy	**Modularity**	**Communities**	**Runtime (s)**
Default Leiden	0.82	79	-
5	0.57	120.59	64.01
8	0.42	18498.83	20.29
7	0.42	18400.19	34.12
0	0.42	18500.65	19.2
3	0.28	17.44	72.08
2	0.26	166.02	164.2
4	0.14	9.16	92.43
6	0.13	342.97	20.3
1	0.061	13.56	160.6
9	0.025	2083.97	28.94

Table 16.5: *Merging strategies on k-split of Twitter-graph (k = 100)*

16.4.3 Preliminary evaluation

Figures 16.3 and 16.4 show that the algorithms that worked best for 2-way split do not work as well as the graph split gets larger. The no-algorithm

(algorithm 0) approach no longer works, and after a couple of initial splits stabilises at around 0.07 modularity. The more aggressive approaches appear to be inconsistent with modularity dropping and rising at different rates depending on the value of k. While most of the algorithms follow the same pattern with some variation, the unpredictability of community-wise merge (algorithm 7) may be interesting to look further into. While it has the lowest lows, it also produces the highest highs. It does not dip below 0, however, as that would mean it places every node into its own community. A potential problem may be that it is too aggressive in its merging. It would be interesting to see how it performs with a higher threshold for merging. Additionally, we believed that there is room for improvement for the best algorithms, i.e. 1, 3, 4 and 5. Attempts at improving the merging strategies were made and will be presented during the following stages.

16.5 Improving the merging algorithms

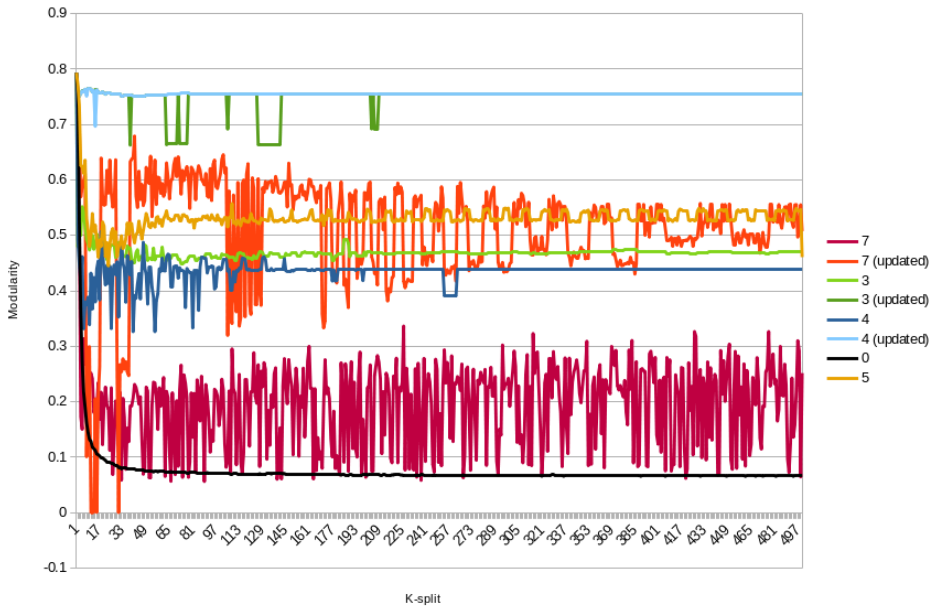


Figure 16.5: *Modularity variations of k=500 with improved merging algorithms on the SBM-graph*

In order to improve the *community-wise merge* (algorithm 7), its main weaknesses were identified. It appears that it was too aggressive in its merging strategy. This fact was supported by the visual representation of the final merge, showing that most nodes in the network were placed in the same community. This confirmed the assumption and the algorithm was modified by introducing a new threshold based on intra-links to inter-links ratio, i.e. the relationship between the edges inside of the cluster and edges between the different clusters. It was discovered that this ratio was high when a community had few intra-links and many inter-links.

In those cases, the concerned communities should be merged and the implementation was changed to accommodate this.

By restricting algorithms 3 and 4 from changing communities in cases of ties, the merging of the SBM-graph seemed to get better. The average modularity loss was about 3 % compared to the non-merging baseline. This finding was deemed to be a promising result. Furthermore, by making small adjustments to algorithm 5, i.e. removing a block for when a node should update its community, the modularity score also seemed to improve.

In order to check the new performance and potential variations of the improved community merge, the algorithm was run with $k = 500$, slicing the SMB-graph in up to 500 partitions, meaning that at $k = 500$, each split consisted of about 8 edges. In order to save time running tests and remove the noise produced by saving results of every k-split, a new variable, `k-step` was implemented in the testing framework. This variable allowed the framework to run the merging algorithm at specific intervals, in stead of running it at each k, e.g. with $k\_step = 50$ and $k = 100$ the merging algorithm would be run three times - at $k = 1$, $k = 50$, and $k = 100$.

Figure 16.5 shows the performance of the updated merging algorithms. The previous results of these algorithms were also included. While still suffering from some variations, *community-wise merge algorithm* (7) no longer dropped to zero once k grew beyond 50. It is also worth mentioning that the other algorithms stabilised at around $k = 150$, with only few occasional variations. This was also the case for the updated ballot-based algorithms (3 and 4). Both seemed to stabilise at about 0.78 as the number of edges in each chunk got smaller. The algorithm 5, presented without the update was shown here for comparison as previously the best best performer to visualise the improvements made by introducing small changes to the algorithms in the refinement phase and its effect. Additionally, algorithm 0 was included to depict the base case.

16.6 Performance of improved merging algorithms

In order to test the newly improved algorithms, the tests were run again, on both the SBM and the Twitter graphs. Both were run with $k = 500$, and the size of the initial paritition clustered by the Leiden algorithm, $\overline{G}_0$, was lowered from 30 % to 20 %. This was done in order to make the merging more challenging for the algorithms. The modularity score was recorded every 25 k, resulting, as previously mentioned, in graphs with less noise, making them easier to read. Figures 16.6 and 16.7 show the modularity scores of all merging algorithms for the SBM and Twitter graphs at each k-split.

The best performing algorithms on the SBM graph were 3, 4 and 5. On the Twitter graph, however only two algorithms (5 and 7) had better scores than the base algorithm (0). The Neighbourhood-to-community link counting (algorithm 5) seemed to work well with the real-world social media graph and the changes made to it seemed to alleviate the

problem it had with continuously decreasing modularity score, as it seemed to stabilise at the score of around 0.75 modularity. Additionally, it is interesting to mention that the Eigenvector Centrality algorithm (9) seemed to perform better as the value of k grew.

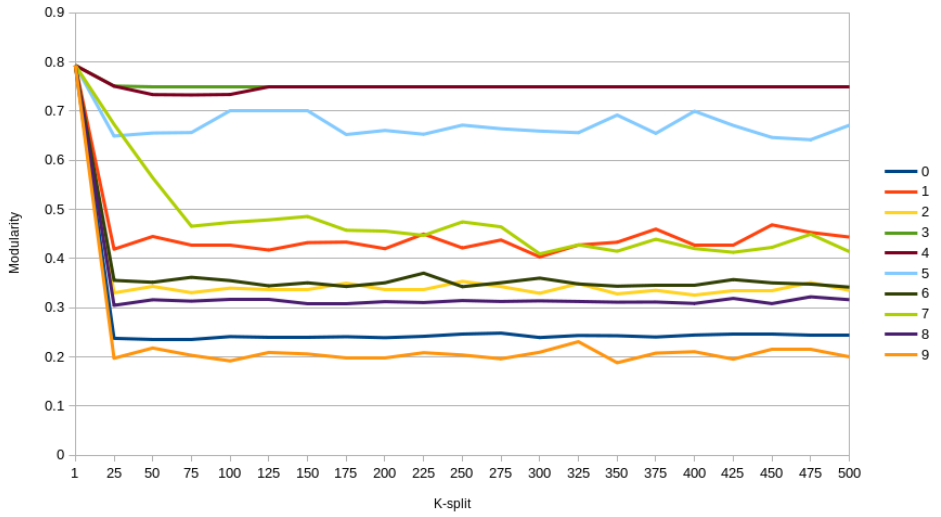


Figure 16.6: *Modularity variations of k=500 with improved merging algorithms on the SBM-graph*

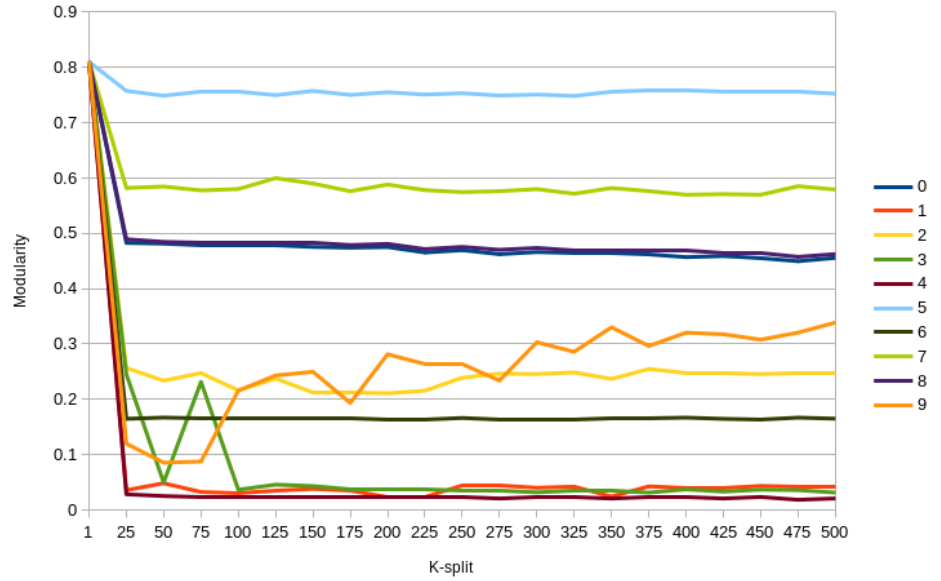


Figure 16.7: *Modularity variations of k=500 with improved merging algorithms on the Twitter-graph*

16.7 Python-igraph and k-split merging

While running the merging algorithm, we noticed that running time of the base algorithm seemed to increase as k grew larger. While testing, we found that when Python-igraph adds new edges, it re-indexes all edges of the graph. The effect of this is that the first merge in a k=200 k-split took 0.1 seconds, while the last one took 2.1 seconds. This happened due to the fact that Python-igraph reprocesses edges that have already been processed before. This directly contradicts the requirement of the running time being proportional to P_k.

In order to circumvent this, changes had to be made in how the the rebuilding of graphs worked. By saving additional information about the neighbours of each node, and not relying on the Python-igraph implementation, the running time was cut down to a constant factor, proportional to the number of edges of P_k, i.e. as long as the number of edges in P_k was constant, the runtime of merging the graph would be constant as well. By storing the information about nodes in Python sets, the running time was decreased even further. Unfortunately, the lookup time for sets is $O(n)$, meaning that the algorithms relying on the information about a node's neighbours for the refinement phase, have a running time of $O(nm)$ where n are the incoming nodes and m are these nodes' neighbours. However, for the purpose of this thesis, this was deemed to be acceptable and in accordance with the requirements, as preliminary tests (not presented in this thesis) showed that the average running time was not increased by much.

16.8 Conclusion

The tests in this iteration were mainly focused on three graphs, i.e. Zachary's Karate Club, the SBM-graph and the SNAP Twitter graph, with the two latter ones being the main focus when transitioning from 2-split to k-split. A testing framework was developed for this purpose and was initially run on the 2-way split, but was later extended to be used for the k-way split, where the user could input the desired graph, in addition to some information about the desired testing patterns.

Additionally, several approaches were introduced and tested. The discovery that different variations of the *Base algorithm* (0) have no real effect was surprising and interesting in equal measures and should be tested further for possible ways to improve it. This, however, is deemed to be out of scope for this thesis, as the *Base algorithm* provides an adequate starting point, and the most important part of the algorithm transpires in the Refinement phase. While several methods presented in this iterations produced good results in 2-split phase, the rapid decline in modularity was somewhat disheartening to see as the value of k increased. The algorithm that appeared to have the best results overall was the *Neighbourhood community links* (algorithm 5), performing above average for the SBM-graph, and being the best for the real-life Twitter-graph, with almost no loss

in modularity. Further testing is needed to make sure that this approach can be used on different graphs and the next iterations will focus on this.

Another interesting finding was the huge variation in performance of the algorithms based on the type of graph being processed. It appears that the graph properties impact the merging and it may be interesting to see which properties have the most influence on the way the merging algorithms perform. Further tests on this will be presented during the evaluation of the merging strategy.

Chapter 17

The NCLiC Algorithm

17.1 Introduction

The merging algorithm number 5, Neighbourhood-to-Community Link Counting, hereby **NCLiC**, was the best overall performing merging strategy out of those tested in Chapter 16. NCLiC is based on the notion that if most of a node's neighbourhood belongs to one community, there is a high probability of that the node itself should be a part of the same community. The algorithm also relies on the guarantees provided by the Leiden algorithm that runs underneath it i.e. that the communities that Leiden provides are optimal, or at least close to optimal. Due to the incremental nature of the graph, and based on the order and composition of the chunks, it is possible that the algorithm provides optimal clusters of the graph. This can occur when the optimal local partitions are also globally optimal, which is the case when the input chunks are the clusters Leiden would produce. This, however, happens rarely and is a best-case scenario. In the other cases, the merging algorithm should perform the merge of the two parts in such a way that the modularity loss is as small as possible, compared to the modularity that would be achieved by clustering the graph as a whole by an off-line algorithm.

Algorithm 17.1 NCLiC refinement phase

```
for node of P_k do
    neighbours ← get_neighbours[node]
    neighbour_votes ← {}
    for neighbour of neighbours do
        neighbour_votes[community_of_neighbour] ← +1
    end for
    new_community ← max(neighbour_votes)
    node_community[node] ← new_community
end for
```

The pseudo-code for the algorithm is presented in **Algorithm 17.1**. After the initial clustering and merging of P_k and $\overline{G}_{k-1}$, transpiring in Phase 1, the neighbours of each node, V_{Pk}, "vote" on which community that node

should be in. The community that receives most votes, *new_community*, is chosen and the node switches its community to the chosen one. In case of ties, a random community is chosen out of the ones that are tied. The algorithm relies on the Leiden producing good clusters and the notion that nodes which are similar will often belong to the same community and thus often be neighbours. The fact that the voting is done for each node in P_k will likely fix eventual suboptimal choices that the initial clustering algorithm may make.

17.2 Steps of NCLiC

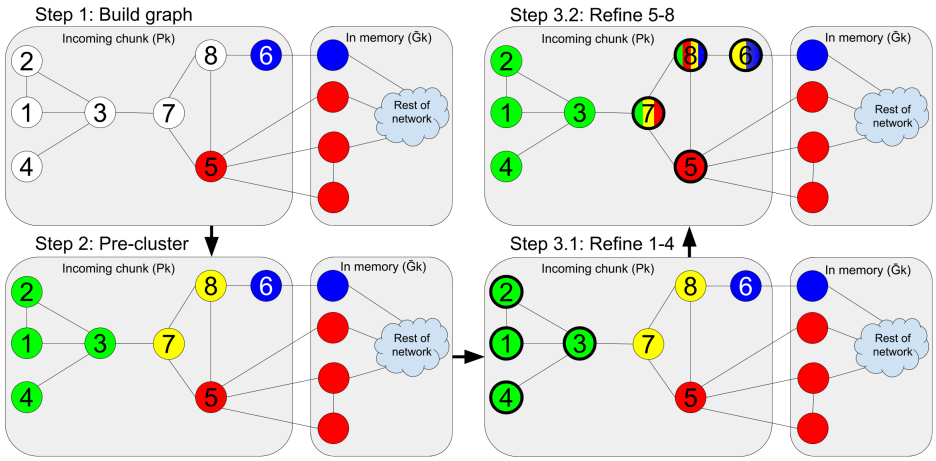


Figure 17.1: *NCLiC Algorithm Steps*

Figure 17.1 shows an example of how the NCLiC algorithm works. The two gray boxes in each step represent the incoming chunk, P_k, (left) and the graph stored in the memory, $\overline{G}_{k-1}$ (right).

Step 1: Build graph P_k is read and the graph of the chunk (G_{Pk}) is built. Nodes 5 and 6 have been processed previously, during processing of some partition, P_{k-y} and exist in $\overline{G}_{k-1}$, and are already placed in clusters, red and blue respectively.

Step 2: Pre-cluster P_k is processed by a chosen algorithm (Leiden) and vertices that are in P_k but not in $\overline{G}_{k-1}$ are added to their discovered communities. In this case nodes 5 and 6 are do not change communities as they have been clustered during a previous merge. The arrows show the progression of the algorithm from one state to another, showing major events. Following is the explanation of each step:

Step 3.1: Refine nodes 1-4 The nodes are iteratively processed and node is moved to the cluster that the most neighbours are a part of. In this case, nodes 1-4 are not moved and retain their cluster. The reason for splitting

up the refinement step in two is to show how the clusters that have been stabilised will stay stable until the graph topology changes in a way that forces a period of instability.

Step 3.2: Refine nodes 5-8 In this step several nodes (5 - 8) are processed:

- Node 5: node number 5 would be moved to the red cluster, due to the number of neighbours in the red cluster (3 neighbours in the red cluster), but is already in the red cluster and stays there.
- Node 6: there is a tie between yellow and blue (node 8 in the yellow cluster and one node, from the memory, in the blue one). Node 6 can be moved to either, chosen randomly.
- Node 7: there is a three-way tie between green, yellow and red clusters (1 node in each). The movement is chosen randomly.
- Node 8: there are several possibilities for the movement of this nodes based on how the nodes were moved prior to this:
  - Red from nodes 5, 7 or both.
  - Yellow from nodes 6, 7 or both.
  - Green from node 7
  - Blue from node 6

The example above shows both strengths and weaknesses of the algorithm. As seen, the more neighbours a node has from the same cluster, the higher the probability of that node being stable, i.e. not changing this cluster. Due to the randomness of the order in which nodes are processed the results may vary. Depending on chance, the entire yellow cluster in the example may be decomposed into separate clusters, which is locally sub-optimal. However, same nodes are often processed several times as new chunks P_k of a graph G are received, potentially resulting in more optimal clustering and often converging towards a stable state, that may be close to an optimal global solution. The example of how nodes can be updated is seen with nodes 5 and 6. An important part of the algorithm is that this can be achieved without the algorithm ever having access to the whole graph, G. Once the clusters have been stabilised, they should remain stable, unless some major disruptions in the network happen, like a new node becoming extremely popular etc. This would initiate a new state of "imbalance", where nodes will shift for a while, and then stabilise again at some point.

17.3 NCLiC Example

In order to describe NCLiC even further, consider the following example. The network presented is a constructed graph of 18 nodes and 34 edges. NCLiC is run on $k = 4$ partitions, with the initial partition size of 20 %. The example will walk through the events as new chunks are received.

17.3.1 Whole graph

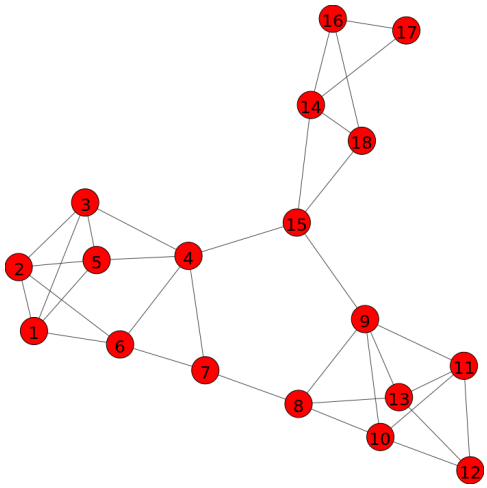
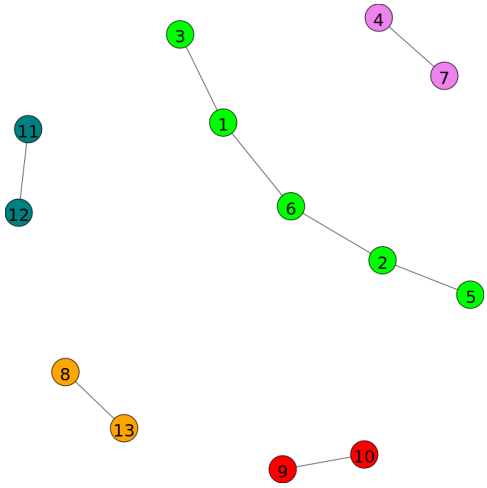


Figure 17.2: *The graph to be clustered by NCLiC*

The graph used in the following example is has a clear community structure, with three clusters. Figure 17.2 shows G before splitting, merging or clustering. The following steps will describe the steps of the NCLiC algorithm as P_k arrive and get merged with $\overline{G}_{k-1}$.

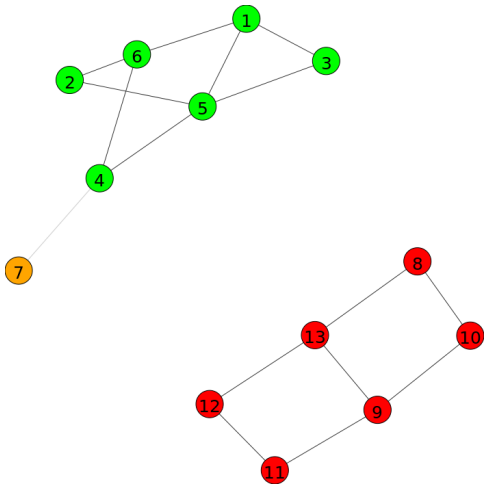
17.3.2 Initial partition



P_1 consists of following nodes: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13].
As the P_1 is received, and there is no graph in the memory, i.e. $\overline{G}_0 = \emptyset$, the first step is to cluster P_1 using the Leiden algorithm. The image shows the result of this clustering. The 13 nodes have been placed in five clusters. No merging algorithm is run during the initial clustering of P_1 as, at best, this would produce the same or worse results than what is already obtained by

running the Leiden algorithm. The information about the clustering and neighbours of each node is stored in memory.

17.3.3 2nd partition

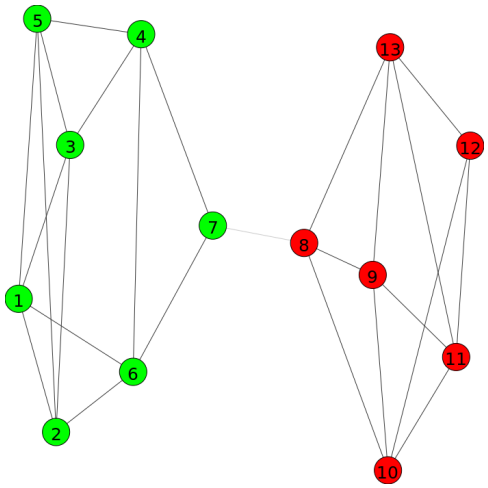


P_2 consists of following nodes: [6, 4, 8, 10, 5, 3, 13, 9, 1, 11, 12].
As the P_2 is received, it is read and the data about each node's neighbours is stored in memory. Then, phase one of NCLiC starts by running the Leiden algorithm on the chunk. Clustering information is updated or added as needed. In this case, the number of nodes in P_2 is less than in $\overline{G}_1$, so only the new nodes, that are not already in the graph are added to the clustering dictionary. The algorithm then progresses to the next phase, where the new nodes' communities are updated. The following nodes change their community from what is stored in the memory:

- Node 4: from pink to green cluster.
- Node 8: from orange to red cluster.
- Node 13: from orange to red cluster.
- Node 11: from teal to red cluster.
- Node 12: from teal to red cluster

Since node 7 was not in P_2, is is not processed, and stays in its original cluster. The reason for it changing colour is that before terminating, in order to save memory, and due to how Python-igraph works with graphs, the algorithm moves node 7 to the lowest number available cluster ID, here number 3, which is marked with orange colour.

17.3.4 3rd partition



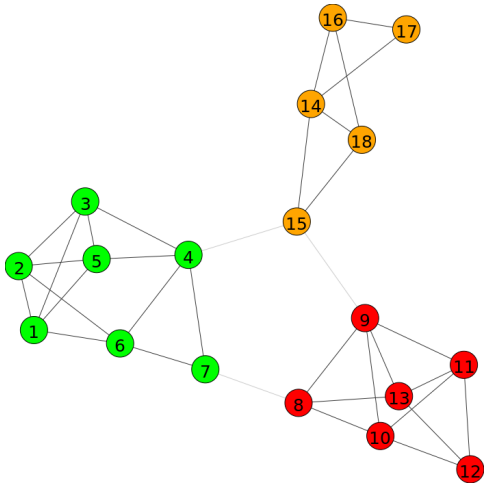
P_3 consists of following nodes: [9, 8, 13, 11, 2, 3, 7, 6, 4, 10, 1, 12].

During the Pre-clustering phase, P_3 gets clustered by Leiden. After the initial clustering, the following nodes are moved between clusters:

- Node 7: from orange to green cluster.

Since no new nodes were added and P_3 was smaller than $\overline{G}_2$, the Leiden algorithm did not make any changes to the intermediate clustering. In phase 2, however, the additional edges that were added resulted in node 7 changing communities, leading to the optimal clustering of $\overline{G}_3$, with two clearly defined communities.

17.3.5 Final partition



P_4 consists of following nodes: [14, 15, 16, 17, 18, 4, 9].

P_4 consists almost exclusively of new nodes, meaning that the nodes 14, 16, 17, and 18 were placed in the new (orange) community. Node 15 was

placed in the pink community, together with 4 and 9 by the Leiden pre-clustering. Nodes 4 and 9, however did not change communities, leaving 15 alone in its community. Intuitively, this is sub-optimal. During the refinement phase, however, the following nodes changed communities:

- Node 15: from pink to orange cluster.

The fact that the refinement phase was able to correct Leiden's clustering, points to how the algorithm is able to build upon the results of the underlying clustering algorithm, but also improve its sub-optimal decisions. While the algorithm has a very limited access to the graph, it appears to produce good results based on decisions taken semi-locally.

17.3.6 Summary

The example shows how the NCLiC algorithm works globally, while only looking at local problems. The fact that each additional chunk seems to improve the clustering indicates that the solution is sound. Taking into account that, in theory, the information in incremental graphs is boundless, this method should suit this type of problem. Furthermore, the example shows how globally sub-optimal initial clustering can be fixed as new information about the network comes in. Additionally, if a whole cluster gets added as a separate chunk, as done in the example, the cluster will likely be preserved. This shows that using NCLiC, new clusters can appear or be merged with the existing ones, but can also be added on their own when the circumstances are right. For long-term observations of networks, this offers a high degree of flexibility, as shifts in the topology of the network, like nodes gaining or loosing popularity, will be reflected by the clustering.

Chapter 18

Evaluation

18.1 Runtime vs Goodness of Clusters

When comparing different algorithms to each other, one has to take into account the time it takes to run the algorithm versus the produced result. Due to the modularity optimization problems being NP-hard, the algorithms that produce good results often take long time to run, while the algorithms that are fast often perform poorly. When it comes to incremental graphs, we do not have a good way of comparing runtime versus the ability to find good clusters. A possible way of comparing algorithms is to try and place them in the same scenario, i.e. run a clustering algorithm like Leiden each time a new chunk is added. In order to do this, the testing framework was extended to facilitate running Leiden on incremental graphs. When a new graph input is received, the graph in memory gets updated and the Leiden algorithm is re-run on the whole graph. This leads to the Leiden algorithm being run on an increasingly larger network, thus increasing the runtime. In order to make the comparison as fair as possible, the rebuilding of the graph was done in the exact same way as in NCLiC. Additionally, the timing of the algorithm allowed for recording the total time algorithm uses, in addition to only algorithm's clustering time.

For comparison, when NCLiC is run on the SNAP Twitter graph with $k = 20$ and initial partition size of 20 %, it takes 12.7 seconds to run with the resulting modularity score of 0.76. If Leiden is rerun on each new input chunk on the same hardware, it takes 2 minutes and 47 seconds with the resulting modularity of 0.81. The question then becomes; is the additional goodness of results worth the longer wait? If we compare the two results, we see that the merging algorithm produces 94 % of Leiden's modularity in just 7 % of the running time. The answer to the question lies within our goal - if we are concerned about finding the best possible clusters and do not care about the time it takes, then the reclustering may be a possibility. Otherwise, concessions must be made in terms of the goodness of clusters.

18.1.1 Leiden runtime

In order to expand on this example, we tested the runtime of the two algorithms, NCLiC and Incremental Leiden, and looked at the number of operations and the time each uses when merging graphs. The results of both were compared to each other. Since the asymptotic runtime is of Leiden is not stated in the paper [40] and the Python-igraph [72] implementation code is obfuscated, we had to estimate the number of operations it does. For the purpose of this test, we assumed the asymptotic runtime of Leiden to be $O(n \log d)$, where d is the average degree, as stated in [44]. While this may or may not be entirely correct, it gave a basis for comparison between NCLiC and Leiden. In order to calculate the number of operations for Leiden, we inserted an operation counter into the Incremental Leiden algorithm which presented the assumed number of operations the Leiden algorithm would use if it had to recluster the graph each time a new graph partition P_k would be added. This was calculated by summing up the result of $n \log d$ for each partition marge.

18.1.2 NCLiC runtime

The runtime of NCLiC for each partition k is dependent on two factors - the runtime of the clustering algorithm in the *Pre-cluster phase*, and the runtime of the merging algorithm during the *Refinement phase*. Following this, the runtime of NCLiC is:

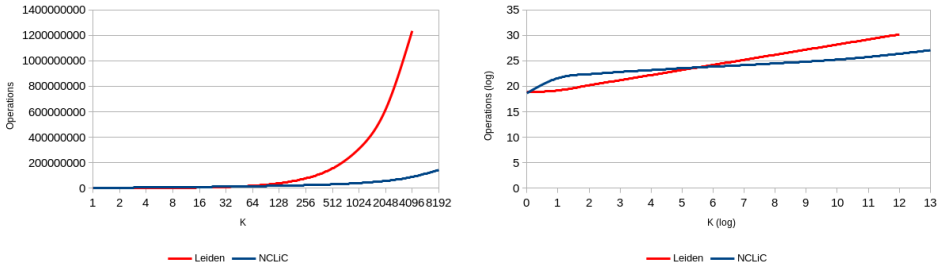
$$O(L_k + n_k d) \tag{18.1}$$

Where L is the runtime of Leiden on partition P_k, n_k is the number of nodes in P_i and d is the number of neighbors of n, which correlates to the average degree d. In order to calculate the number of operations of NCLiC, a counter was added in each of the phases. The Pre-cluster counter was calculated through the use of Leiden's assumed asymptotic time, $n \log k$, while another counter was inserted in the innermost loop of the merging algorithm. The sum of these elements provided the number of operations of the algorithm for a specific k.

18.1.3 Leiden vs NCLiC

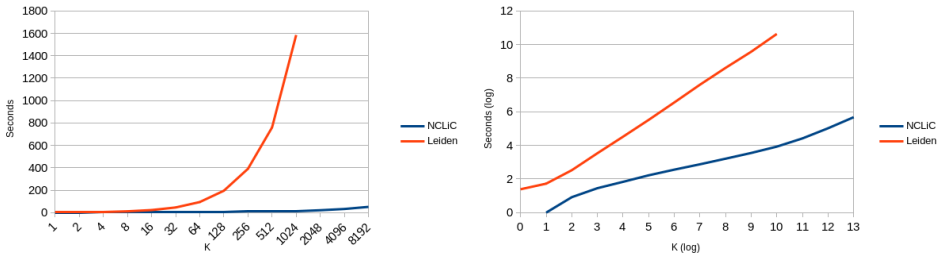
Two types of tests were run in order to establish the running times of the NCLiC and Leiden. The first one, based on the aforementioned calculations was run to determine the growth of the number of operations as the value of k increases. The second one was a timed test - for each algorithm the total amount of time to cluster the required number of k was recorded, in addition to its clustering time only. The timed tests were run on a personal computer, with the same specifications as previously mentioned. The SNAP Twitter graph was used with all tests, with 20 % of G set to be the size of the initial graph, P_0. Each test was rerun up to 13 times with increasingly larger k and the results were recorded. Due to long running

times, the Leiden approach was only run for up to $k = 1024$ in regards to time, and $k = 4096$ in regards to operations.



(a) *Operations* (b) *Log2 Operations*

Figure 18.1: *Performance of NCLiC vs Leiden in number of operations*



(a) *Seconds* (b) *Log2 Seconds*

Figure 18.2: *Performance of NCLiC vs Leiden in seconds*

Figure 18.1 shows the number of operations done by NCLiC and Leiden algorithms, presented in normal (log-num) and logarithmic (log-log) scale. Figure 18.2 shows the running time of the two algorithms in seconds, also presented in normal and logarithmic scale. The runtimes presented are the clustering times only it order to compare the algorithms as fairly as possible. The purpose of this is to account for the deficiencies of the graph reading and rebuilding. The rebuilding of the graph is a separate problem, as is deemed out-of-scope for this thesis. It is, however, important to mention that a well-implemented rebuilding of graphs would make NCLiC even faster. As seen in the Figure 18.2, the time it takes to rebuild the graph takes increasingly longer time the larger k becomes for both algorithms. Due to this fact, it may be difficult to compare the two algorithms by looking at their running times. It is, however, telling that the Leiden reclustering approach was not run for values of k larger than 1024, as the processing times became too large (over 45 minutes). NCLiC accomplished this in just 26 seconds, including the rebuilding time, and 15 seconds, excluding it. It is also worth mentioning that the Leiden algorithm is implemented in C++, which is far superior to NCLiC's Python implementation, in regards to running times. Nonetheless, the processing time of NCLiC is beyond comparison to the approach of continuous reclustering.

In terms of operations, presented in Figure 18.1, the NCLiC and Leiden algorithms start out equally, but the number of operations of NCLiC increases rapidly for the first few k, as the Refinement phase is added. However, the growth of the number of operations of NCLiC is less than that of Leiden and already after $k = 32$ Leiden's operation count grows beyond that of NCLiC and continues to double for each time k doubles. NCLiC's operation increase is far slower, which also results in faster running times. This is also evident in clustering times of the algorithms, where no overlap can be found at all. Again, the increase of Leiden is exponential, while NCLiC's is much less steep.

18.2 Initial partition size

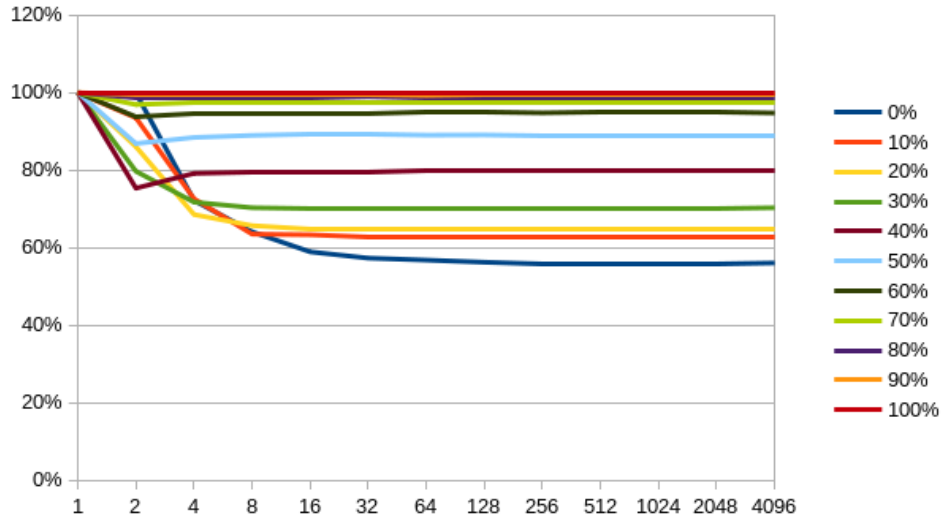


Figure 18.3: *NCLiC performance based on initial partition size*

Due to the fact that the NCLiC does not have access to the information about the whole graph, the size of the initial partition, P_k is extremely important. Figure 18.3 shows an example of how the modularity score of NCLiC depends on the initial partition size of the graph, G. For the purpose of testing, the framework was used with various sizes of the input partition in percent of the total number of edges of the graph - at 0 %, the algorithm starts with no $\overline{G}$ and the thus no clustering information to base its preliminary decisions on. At 100 % the algorithm receives G to cluster once, with no additional P_k to cluster. The result of this is effectively the same as running Leiden once G.

The graph used for this test is the wing.mtx, from the DIMACS10-set. It is a small (10 937 nodes and 150 976 edges), simple graph with Leiden modularity score of over 0.9. The reason it was chosen for these tests was that NCLiC had struggled with this graph in previous tests, and resulted in one of the largest differences in modularity between Leiden and NCLiC (34 %), meaning that it is one of the graphs that was especially hard to cluster

for NCLiC.

The figure shows that the larger the initial partition is, the lower the modularity drop becomes. The figure presents the modularity score as percentage of the total maximum achievable modularity score, obtained by running Leiden. As seen, there is a diminishing return of increasing the initial partition above 50 %. There is also a small difference between the initial partition of 10 % and 20 %. At 50 % of the initial partition size, the modularity stabilises at around 85 % of the attainable score. It is however interesting to note the modularity drop at lower k values, which happens with almost all partition sizes. At $k = 2$, the highest drop in modularity happens at 40 % of the initial graph size and lowest drop happens at 0 % of the initial partition size, while at $k = 4$ and $k = 8$ with 10 % of the initial partition size resulted in better modularity score than 20 % initial size.

This leads to the conclusion that a larger starting graph will result in better performance of NCLiC. While there may be some variations in the beginning, the clusters will stabilise after a while. If it is a possibility, one should start with as much of the graph as possible. This is likely in a streaming scenario, where some data about the graph may have already been gathered and stored in memory. Another possibility is to recluster the whole graph at some time intervals, as this would transform $\overline{G}_k$ into $\overline{G}_0$, leading to higher modularity scores of future merges.

18.3 Merging and Graph Characteristics

In order to determine whether the characteristics of a graph can indicate how the merging algorithm will perform, four different graphs were tested. Their characteristics and the performance of NCLiC were recorded in order to see whether some characteristics could impact the algorithm. Table 18.1 presents these differences. The graphs tested were the SBM, SNAP Twitter, DIMACS10 598a and SNAP YouTube and were merged with $k = 20$ and 20 % initial partition size.

	SBM	**Twitter**	**598a**	**YouTube**
Leiden Modularity	0.79	0.81	0.90	0.73
NCLiC Modularity	0.49	0.77	0.75	0.63
Modularity Retention	62 %	95 %	83 %	86 %
Nodes	1100	81 306	110971	1134890
Edges	4063	1 342 310	741934	2987624
Average degree	7.4	33.0	13.37	5.26
Density	0.0067	0.0004	0.00012	4.64e-06
Neighbour degree	8.3	191.9	13.77	654.11
Eigenvector centrality	0.0109	0.0005	0.00165	0.00017
Average clustering	0.05	0.57	0.43	0.08

Table 18.1: *Characteristics of different graphs*

Looking at the various characteristics of the graphs, there does not

seem to be any clear connection between them and NCLiC's modularity retention. The highest performance of NCLiC was achieved on the SNAP Twitter graph, which also had the highest average degree and average neighbour degree, i.e. the average degree of the neighbourhood of node u [79]. This may have impacted the algorithm, as it had more chances to "fix" the sub-optimal decisions in the pre-clustering phase. Additionally, the graph topology has an impact on how clusters get merged. When the original clusters are large and input chunks are small, Leiden may incorrectly partition nodes that should have been placed in the same cluster. From the local perspective, this partitioning may be true, but in the global perspective, the results would be highly skewed. The fact that there does not seem to be any way of knowing how NCLiC will perform beforehand shows us that NCLiC suffers from the same problem all other clustering algorithms seem to suffer from - namely that you have to choose and adapt the clustering algorithms in accordance with the type of graphs you are working with, as the performance of the algorithms depends on the structure of the graphs.

18.4 NCLiC vs Current State-of-the-art Algorithms

18.4.1 Graph Description

The tests in the following sections were run on a set of six graphs, with various sizes and characteristics. Several of the graphs are from the SNAP-library, as they represent real-world social networks. Additionally, one DIMACS10 graph was selected, as it presented a challenge for NCLiC. The following graphs were used:

Stochastic Block Model The SBM graph, presented in Section 10.5, Benchmark Networks.

SNAP Twitter The Twitter graph used, presented in Section 10.5, Benchmark Networks.

SNAP Email From the description of the Email network from SNAP database: *"The network was generated using email data from a large European research institution. We have anonymized information about all incoming and outgoing email between members of the research institution. There is an edge (u, v) in the network if person u sent person v at least one email. The e-mails only represent communication between institution members (the core), and the dataset does not contain incoming messages from or outgoing messages to the rest of the world"* [80].

SNAP YouTube From the description of the YouTube network from SNAP database: *"Youtube is a video-sharing web site that includes a social network. In the Youtube social network, users form friendship each other and users*

can create groups which other users can join. We consider such user-defined groups as ground-truth communities. This data is provided by Alan Mislove et al" [81].

SNAP Amazon From the descriptiono of the Amazon network from SNAP database: *"Network was collected by crawling Amazon website. It is based on Customers Who Bought This Item Also Bought feature of the Amazon website. If a product i is frequently co-purchased with product j, the graph contains an undirected edge from i to j. Each product category provided by Amazon defines each ground-truth community"* [82].

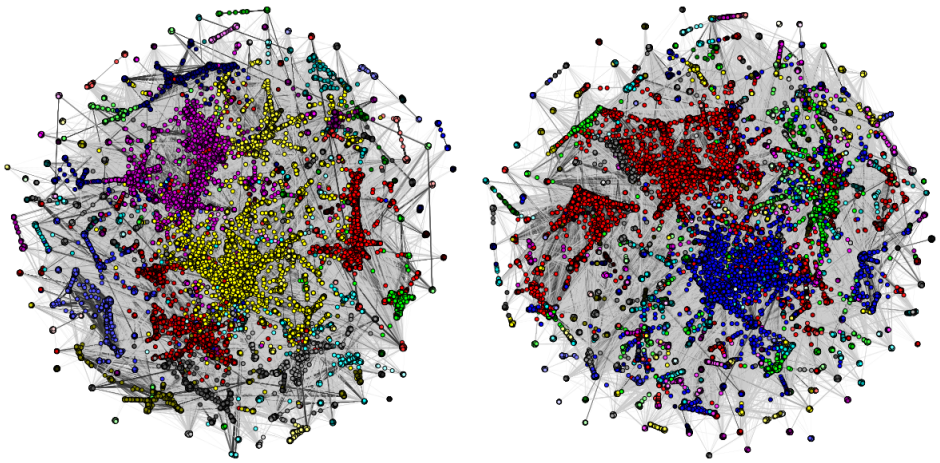
DIMACS10 Wing The Wing network from the DIMACS database. The network consists of 62 000 nodes and 121 500 edges, which is large enough to provide a clustering challenge, but small enough to perform tests fast.

18.4.2 Testing

Due to the lack of other clustering algorithms for incremental graphs and the choice of modularity as the main goodness measure of the clusters, this thesis uses the modularity of Leiden algorithm as the reference. This is done because the Leiden algorithm achieved the highest modularity during the preliminary test. There are, however, many other frequently used clustering algorithms. For this purpose, we tried to compare the NCLiC algorithm with other state-of-the-art algorithms. To make this process easier, we only tested algorithms implemented in Python-igraph. NCLiC was run at $k = 20$ with the initial partition size of 20 %. Table 18.2 shows the results. It is interesting to see how different algorithms perform in terms of runtime, modularity and number of communities discovered. It is however important to mention that the results can be somewhat misleading. The best performing algorithms - Leiden and Louvain - are built for maximizing modularity. It may therefore be unfair to compare the chosen clustering algorithms against each other, as they are all built for different purposes. The implementations of the different algorithms also vary, which heavily impacts the runtime - SCoDA, for example was tested by using the Python implementation, which is by default slower by several magnitudes than its C++-implementation. The Louvain and Leiden algorithms benefited from this the most, as also seen by their runtimes. For the purpose of testing of the NCLiC algorithm, however, there are some interesting results. NCLiC is better than SCoDA across the board, with the exception being the SNAP email-network, where they performed equally. On the larger, real-life graphs (Twitter, YouTube, Amazon), NCLiC even beats Infomap, either by having higher modularity, or by being able to complete the clustering within reasonable time[1]. It is, however, important to mention that while the modularity score is a good way to test for goodness of clustering, the clusters discovered may vary despite algorithms producing similar modularity scores. Figures 18.4 and 18.5 depict an example of the

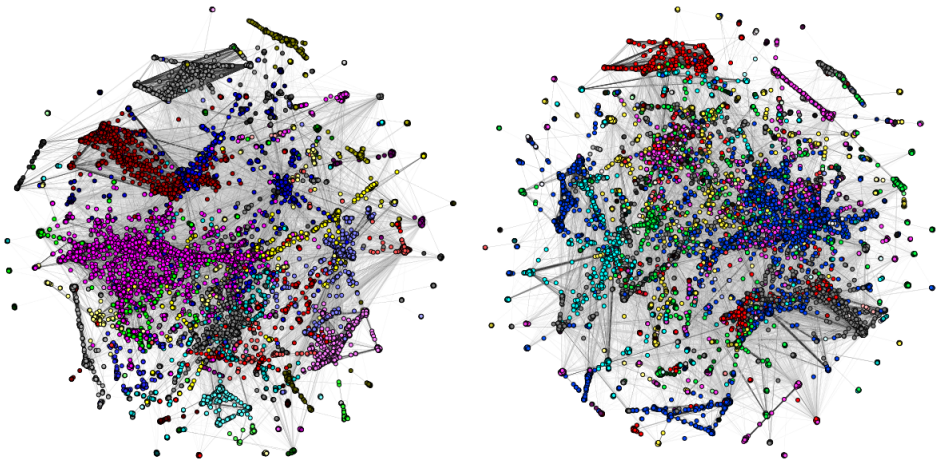
[1]Infomap was stopped after failing to complete clustering of the SNAP-YouTube graph in more than three hours

differences in clustering when using Leiden versus NCLiC. Whether one or the other is more correct is left up to for debate and is out of the scope of this thesis.



(a) *Leiden* (b) *NCLiC*

Figure 18.4: *Twitter clusters discovered by Leiden and NCLiC*



(a) *Leiden* (b) *NCLiC*

Figure 18.5: *Facebook clusters discovered by Leiden and NCLiC*

Graph	Algorithm	Modularity	Communities	Time
SBM				
	NCLiC	0.45	82	0.02 s
	Leiden	0.79	7	0.008 s
	Louvain	0.79	7	0.009 s
	Infomap	0.79	8	1.12 s
	SCoDA	0.22	247	0.006 s
SNAP Twitter				
	NCLiC	0.75	634	11.21 s
	Leiden	0.81	73	1.52 s
	Louvain	0.79	79	2.02 s
	Infomap	0.43	2914	4 m 21 s
	SCoDA	0.05	41 441	7.01 s
SNAP Email				
	NCLiC	0.08	32	0.19 s
	Leiden	0.43	27	0.006 s
	Louvain	0.42	27	0.016 s
	Infomap	0.41	42	0.30 s
	SCoDA	0.07	492	0.02 s
SNAP YouTube				
	NCLiC	0.63	72 534	38.81 s
	Leiden	0.72	7 272	16.39 s
	Louvain	0.68	9 709	12.29 s
	Infomap	-	-	> 3 hours
	SCoDA	0.14	724 495	14.42 s
SNAP Amazon				
	NCLiC	0.77	26 784	12.61 s
	Leiden	0.93	393	3.39 s
	Louvain	0.92	261	3.67 s
	Infomap	0.82	17 285	24 m 34 s
	SCoDA	0.36	104 875	3.42 s
DIMACS Wing				
	NCLiC	0.59	8 545	1.55 s
	Leiden	0.91	41	0.34 s
	Louvain	0.90	43	0.59 s
	Infomap	0.76	2 108	11 m 15 s
	SCoDA	0.45	10 988	0.45 s

Table 18.2: *State-of-the-art algorithms and NCLiC*

18.5 NCLiC with other clustering base algorithms

As previously mentioned, Leiden was used as the base clustering algorithm in NCLiC. It may, however, be exchanged with other clustering algorithms. For this purpose, tests were run in order to determine the performance of NCLiC with different clustering algorithm. The tests were run with $k = 1024$ and initial partition size of 20%. The initial partition was clustered by Leiden for all tests. Figure 18.3 shows how graphs processed in the the previous section performed, if the base clustering algorithm was exchanged. The algorithms used were Leiden, Louvain and Infomap, in addition to running no clustering algorithm. The way this was accomplished, was by removing the clustering of the incoming chunk, and in stead giving each new node its own cluster.

The results appear to be similar between the different base clustering algorithms. Both modularity score and the number of communities found seem to be the roughly the same, no matter which clustering algorithm was used. A notable exception is when running NCLiC with no clustering algorithm, the results are slightly to noticeably better. While the modularity score of *No pre-clustering algorithm* is slightly higher across the board, the number of communities found is significantly lower, pointing to the fact that the merging of clusters is more aggressive in this approach. This discovery is surprising, as the NCLiC algorithm was designed around the premise that the clusters it receives from the chosen clustering algorithm are important. The test, however, shows that the choice of the algorithm may not be as important as initially thought, and that NCLiC does not actually need to cluster the chunks beforehand. This also speeds up the process, as an entire step of Leiden clustering may be removed (in addition to several supporting actions required by the Python-igraph framework). The significance of this discovery will be tested later in this thesis.

It is also noteworthy that the algorithm was able to run Infomap on the SNAP YouTube graph in just over 4 minutes, when it previously stalled out after over 3 hours of processing time! This leads to a possibility of running slower algorithms by using NCLiC as an intermediary.

Graph	Algorithm	Modularity	Communities	Time
SBM				
	-	0.51	65	0.15 s
	Leiden	0.48	101	0.18 s
	Louvain	0.42	99	0.18 s
	Infomap	0.46	103	0.26 s
SNAP Twitter				
	-	0.78	645	27.87 s
	Leiden	0.76	935	26.61 s
	Louvain	0.75	943	29.53 s
	Infomap	0.76	1088	1 m 22 s

SNAP Email				
	-	0.08	35	0.75 s
	Leiden	0.07	33	1.89 s
	Louvain	0.04	33	0.95 s
	Infomap	0.04	36	1.79 s
SNAP YouTube				
	-	0.65	51 791	55.84 s
	Leiden	0.62	97 345	59.34 s
	Louvain	0.62	97 598	57.29 s
	Infomap	0.63	99 141	4 m 13 s
SNAP Amazon				
	-	0.84	20 059	18.07 s
	Leiden	0.78	28 899	58.07 s
	Louvain	0.78	29 114	58.95 s
	Infomap	0.78	28 937	1 m 30 s
DIMACS Wing				
	-	0.62	7 045	2.36 s
	Leiden	0.59	8 732	5.87 s
	Louvain	0.59	8 760	5.94 s
	Infomap	0.59	8 754	9.07 s

Table 18.3: *NCLiC performance with different clustering algorithms*

18.6 Shuffled versus Non-shuffled Graphs

The discovery of NCLiC performing better without a dedicated clustering algorithm presented in previous section prompted us to conduct further testing. Until now, most tests have been conducted on edge lists as-is, provided by the source. These are often (but now always) sorted in some way, for example listing all edges from one node before moving on to the next one. Due to the fact that the NCLiC algorithm is dependent on the order in which new edges are added, this method has been chosen in order to not give any potential advantages or disadvantages to specific runs, where the shuffling may have skewed the results. Table 18.4 shows the difference in performance of NCLiC on shuffled and non-shuffled variations of graphs. For the purpose of these tests, NCLiC was reverted to use Leiden as the base clustering algorithm.

Graph	**Algorithm**	**Modularity**	**Communities**	**Time**
SBM				
	Non-shuffled	0.51	65	0.15 s
	Shuffled	0.67	25	0.14 s
Twitter				

	Non-shuffled	0.78	645	27.87 s
	Shuffled	0.81	228	2 m 46 s
Email				
	Non-shuffled	0.08	35	0.75 s
	Shuffled	0.003	32	0.75 s
YouTube				
	Non-shuffled	0.65	51 791	55.84 s
	Shuffled	0.66	106 603	5 m 59 s
Amazon				
	Non-shuffled	0.84	20 059	18.07 s
	Shuffled	0.80	26 274	29.15 s
Wing				
	Non-shuffled	0.62	7 045	2.36 s
	Shuffled	0.53	10 299	2.25 s

Table 18.4: *NCLiC on shuffled vs non-shuffled edge lists*

The results do not conclusively indicate that the shuffling the graphs leads to higher modularity. The results between shuffled and non-shuffled graphs vary to some degree, and the performance appears in some cases to be better for non-shuffled graphs, while better for shuffled in other instances. A discovery worth mentioning is that the shuffled graphs seem to have a longer runtimes. Some quick testing seems to indicate that this is due to the fact that more individual nodes are added simultaneously through each chunk, meaning that the algorithm has more neighbours to check. On average, about 20 % more neighbours were checked when the graphs were randomly shuffled, than when using the original order of the edge-lists. This may also be the reason for occasional performances increase produced by the algorithm on the variations with shuffled edges.

In streaming problems, every ordering of the incoming data is a separate problem. In processing of incremental graphs, every ordering of chunks is. Discovering the potential optimal order would be an interesting problem. However, due to the fact that the performance was not consistently better across all graphs, this was deemed out of scope for this thesis.

18.7 Weaknesses

The main weakness of NCLiC is that the runtime grows primarily with the number of neighbours of nodes in each chunk. The algorithm will run into performance problems if a dense network continues to grow towards infinity. This limitation may be somewhat alleviated by probing the neighbours; i.e. when the number of neighbours becomes too large, a randomly chosen portion of these can be checked and the chosen

community extrapolated.

Another apparent problem of NCLiC is a problem that is common amongst most of the clustering algorithms - the performance is extremely dependent on the type of graphs that is being processed. The topology of the graph dictates whether an algorithm will work on that graph or not. This is as true for NCLiC as it is for the state-of-the-art algorithms presented in this thesis.

Chapter 19

Large Scale Testing

19.1 K-merge on DIMACS10 (Leiden pre-clustering)

19.1.1 Implementation

In order to test the NCLiC merging algorithm on a larger scale, tests were run on the DIMACS10 graph set. NCLiC, with Leiden as the base pre-clustering algorithm, was run on 147 of the available 151 graphs. The graphs that were larger than 900 MB, which translated to 108 million edges or more, were not processed. The tests were run on the Simula Ex[3] super computer. The k-split was set to increase by powers of two in order to be able to detect small changes in the beginning, when k is small and detect trends as k grows. The graphs were split into two groups based on their size, where graphs under 50 MB were run until $k = 2^{12}$, while graphs between 50 MB and 900 MB were run up to $k = 2^{10}$. Additionally, all graphs had an initial partition size of 20 %, meaning that the original graph of the first run of every k-split had 20 % of all edges in the edge list. The remaining 80 % of the edges were split into k parts. In order to make this experiment reproducible, the graphs were not shuffled.

19.1.2 DIMACS10

The processed graphs vary widely in size. The smallest graph had 39 nodes and 340 edges, while the largest had over 16 million nodes and over 100 million edges. Additionally, the set consisted of weighted, random, normal and multigraphs. Multigraphs are graphs that can have several edges shared between the same pair of nodes. All graphs are undirected.

The graphs generally had high modularity when clustered with Leiden - with an average modularity of 0.89. 112 graphs had a Leiden modularity score over 0.90 and just 14 graphs had Leiden modularity of under 0.6. 21 graphs had a modularity score between 0.6 and 0.9. The graphs with the lowest modularity were multigraphs, which had an average modularity of 0.06. Due to the fact that the NCLiC algorithm is not meant to be used on multigraphs, these were removed from the results.

19.1.3 Results

The average performance of NCLiC in terms of modularity was 0.76. This corresponds to about 83 % modularity retention compared to the Leiden modularity score. The graphs that had the highest drop in modularity were the multigraphs, with around 98 % drop. While this may appear significant, the low starting modularity score results in an actual drop of around 0.06 points of modularity. It is also worth mentioning that 99 of 142 graphs had a modularity drop of less than 20 %. Figure 19.1 depicts the distribution of the graphs based on merged modularity retention. The bars represent the number of graphs, while the line represents the corresponding distribution in percent, split into 10 % bins. As seen, 93 % of graphs had modularity retention of 70 % or higher. Additionally, 26 graphs (18 %) had retention of 90 - 100 % .

Figure 19.2 shows change in modularity as k is increased, where Figure 19.2a shows the non-normalized numbers of the groups, while 19.2b shows the modularity score as percent of the average score of the Leiden algorithm. The graphs are partitioned into two groups - presented as *huge graphs* and *medium graphs*. Graphs with the count of edges between 10 million and 100 million are considered huge. The rest of the set was called medium do the the average number of edges in this set being 1.2 million. The graphs in this set were up to 10 million edges. Additionally, an average of the two groups is shown.

In figure 19.2a we can see that there seems to be little difference between the performance of NCLiC on huge and medium graphs, with modularity stabilising at about modularity of 0.8 at $k = 4$. The normalized figure shows that the huge graphs have the highest drop in modularity initially, but looking to improve slightly as k increases. In general terms, however, it does not seem that the size of the graph has any significant impact on modularity retention with average retention of huge graphs being 84 % and for medium - 82 %.

Looking closer at the performance of NCLiC, an interesting discovery is that all of the Open Street Map (OSM) graphs, presented in DIMACS10 set, have low modularity loss (between 0 % and 10 %). These graphs are based on real road networks in countries and regions, which have high diameter. On the other hand, the graphs that NCLiC had difficulties with were of a synthetic nature. It is important to mention that the DIMACS graphs present two types of problems - partitioning and clustering. It is therefore encouraging to see that the algorithm had a relatively stable performance on both types.

19.2 K-merge on DIMACS10 (No pre-clustering)

19.2.1 Implementation

The discovery about the potential benefits of removing the clustering algorithm in the pre-clustering phase of NCLiC, presented in Chapter 18 lead to the decision of running a separate test for the NCLiC algorithm

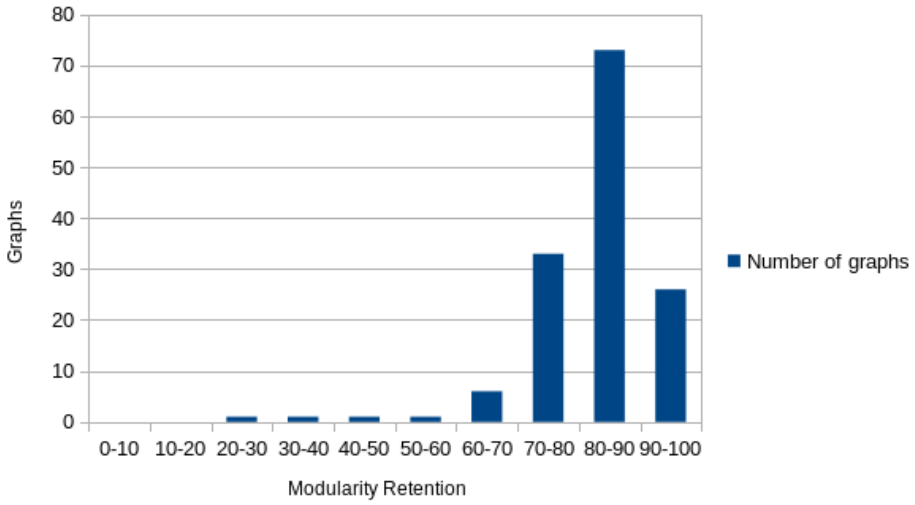
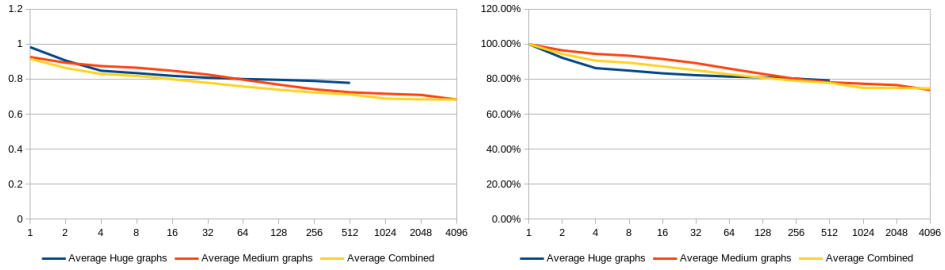


Figure 19.1: *Modularity retention count*



(a) *Non-normalized* (b) *Normalised*

Figure 19.2: *Performance of NCLiC vs Leiden as k grows on SNAP Twitter Graph*

with no pre-clustering. Apart from this change, the tests were identical to the ones described above. The multigraphs were again removed from consideration when running the tests and analysing the results.

19.2.2 Results

Figure 19.3 shows the distribution of modularity retention. The average modularity of the clustering was 0.68, which corresponds to 74 % modularity retention. Also here, the graph sizes did not seem to impact the modularity score significantly, with huge graphs having 72 % modularity retention and medium graphs - 75 %. The average modularity retention was 9 % lower than what was achieved with its Leiden-based counterpart, which is significant. Additionally, fewer graphs had modularity retention of 80 - 90 %, and the largest portion of graphs (over 40 %) was now in the 70 - 80 % range. It appears that the results achieved in Chapter 18, were not representative for the general performance of NCLiC. Figure 19.4 depicts the performance as k is increased. It is interesting to see that the performance of this variation of NCLiC seems to become better with

a higher number of partitions. The average modularity retention score between $k = 2$ and $k = 4096$ is 0.7 (8 %), which indicates that processing the graph in many small partitions can yield better results compared to few large ones. While the variation did not produce a better result, these tests serve to show the importance of trying out different clustering algorithms used in the initial clustering phase of NCLiC. A possibility may also be to add a switch based on the size of the input, running *"Leiden-based NCLiC"* in the beginning to provide a good clustering base, while later switching to *"No-base NCLiC"* providing the "self-fixing" benefits in the later stages of the run. This, however, is deemed out of scope for this thesis.

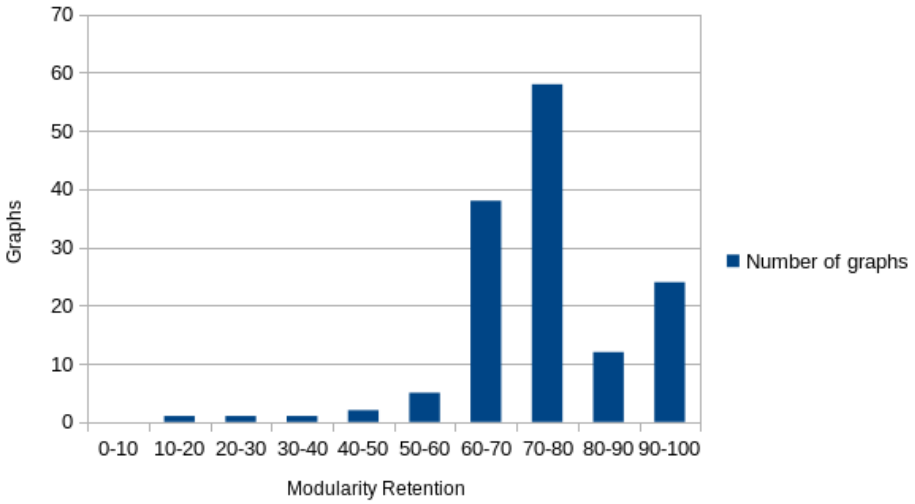
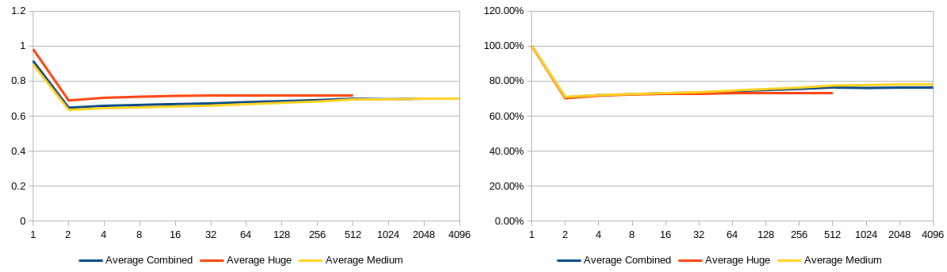


Figure 19.3: *Modularity retention count, no clustering algorithm*



(a) *Non-normalized* (b) *Normalised*

Figure 19.4: *Performance of NCLiC vs Leiden as k grows on SNAP Twitter Graph, no clustering algorithm*

Chapter 20

Contribution

This thesis has focused on community detection in incremental graphs. The following topics were covered in this thesis.

The incremental nature of the graphs makes the current state-of-the-art algorithms unviable, as it lies in the intersection of the *regular* and *stream-based* clustering domains. This thesis has proposed a set of requirements that need to be fulfilled in order to be successful in working with incremental graphs. Based on these requirements, a novel approach has been presented where a graph gets continuously updated in runtime that is proportional to the size of the incoming partition of the graph. This makes the merging feasible in a semi-streaming (on-line) scenario as presented in the problem.

20.1 Testing of state-of-the-art algorithms

Several state-of-art algorithms were presented and evaluated in the thesis, among these were the Leiden, Louvain, InfoMap, and SCoDA algorithms. These algorithms were adapted and tested in an off-line and on-line environments. Based on their performance, two algorithms, namely SCoDA and Louvain, were chosen as candidates for further research, with Louvain being an offline algorithm, while SCoDA - a streaming algorithm. The two were extensively tested on nearly 6000 graphs and their performance was recorded and evaluated.

20.2 Introduction of incremental graphs

In order to solve the problem at hand, encountered in the UMOD project, a new definition of graphs was introduced. The implications of working on these types of graphs, which are built incrementally, by adding new partitions of the graph, were called *incremental graphs*, which, being in the intersection between off-line and on-line domains, introduced several novel problems, not covered in the literature currently available. A novel take on using Leiden in a streaming scenario, which consisted of merging the two state-of-art algorithms, *SCoDA* and *Leiden*, called **SCODA-Leiden**

was proposed. While this algorithm performed well, its worst-case runtime was deemed sub-optimal, and further work on it was abandoned. This lead to the establishment of a set of requirements for working with incremental graphs. Additionally, this lead to the notion that a possible approach for working with incremental graphs, is to design an algorithm that is able to merge the incoming partitions in a way that leads to a minimal loss of good clusters.

20.3 Testing framework for incremental graphs

During the course of this thesis there was a consistent need of comparing different merging algorithms. A testing framework was developed for that purpose, capable of splitting and merging graphs and their clusters, called the **k-split merge**. Ten different merging techniques were tested. Additional improvements were made to the initial approaches that lead to better results. The framework was used and extended to cover the necessary demands for information. The implementation and results of the thorough testing of these, and other algorithms, on a number of different graphs is described and presented in this thesis.

20.4 Incremental graph clustering algorithm

The main purpose of this thesis was to create a clustering algorithm capable of clustering huge graphs with satisfactory results, while adhering to the requirements of clustering incremental graphs. The different iterations presented in this thesis lead to a concluding result of a novel incremental clustering algorithm, called **NCLiC**, which was tested on over 150 different graphs. The tests showed that the algorithm's runtime is unrivaled to the alternative solution of reclustering the graphs as new partitions are received. Its clustering performance is also very good, with 83 % average modularity retention, and over 95 % on some types of graphs. In many cases it even performs better than the current state-of-art streaming algorithms like SCoDA, and regular approaches like InfoMap and Louvain.

Chapter 21

Future Work

21.1 Merging Algorithm

21.1.1 Implementation of Algorithms

During this thesis, the Python-igraph library has been primarily used as the base for creating algorithms. While powerful and allowing for many possibilities, it has some restrictions and problems attached to it. For a possible way forward, a separate implementation should be considered, not bound to Python-igraph. This would allow to streamline algorithms to their primary purpose, making them faster and more adapted to solving the problem.

Another part of this is to create the algorithms in another language than Python. While it is extremely useful for prototyping and testing, there are many other programming languages that are faster. A possible solution may be to use C or C++, and several algorithms mentioned in this thesis have a C++ implementation. While testing different algorithms, the SCoDA C++ implementation was tested against its Python variant and it was discovered that Python was over 50 % slower. This, however may be due to differences in implementation, as Python language is considered to be about 400 times slower than C++.

21.1.2 Using Leiden by contracting network

When developing the algorithms during this thesis, the main focus was placed on the edge- and node-wise merging of graphs. Another possible way of doing this may be by contracting graphs, i.e. treating discovered clusters as nodes. Doing this may lead to increased processing speeds. It is unknown whether this may provide better clustering results, and is therefore not been a priority in this thesis. It should, however, be tested further.

21.1.3 Weighted edges

The NCLiC algorithm designed in this thesis does not take into account weights of a weighted graph. However, it is extensible, and can be adapted

with ease. I would argue that extending NCLiC to take into account possible weights would make it perform even better, as it would have more information about the importance of nodes. Modifying the link counting to include weights would make the decision of changing communities of each node easier and can be achieved by introducing a weight variable for neighbouring nodes. By doing this in a smart way, for example by hashing node ids and their weight, the performance would likely be even better, reducing the reading of neighbours to linear time. Due to the fact that this could be easily implemented and would potentially provide major benefits to the clustering results, this should be one of the main priorities in the case of further work with the NCLiC algorithm.

21.1.4 Additional network modifications

Many real-life networks are, as previously mentioned, dynamic by nature. Usually, new nodes appear and form links to other nodes, but nodes and links can also disappear. When looking at social media, this phenomenon of graph rewiring occurs quite often. This can happen as users unfollow previously followed users or topics and discover new ones. Users can delete their accounts or get removed in some way, and on-line clustering algorithms must accommodate these changes. The same is also true for incremental graph-algorithms, which can receive different information in the input partitions. The NCLiC algorithm in its current form does not provide a way to cluster based on deletion of edges and/or nodes. A possible way forward is therefore to implement this feature. This would however mean, that the notions of deletion of edges and nodes must be implemented in the algorithm.

In circumstances where graphs can evolve by both growing and shrinking, the evolution can be split up into the following four tasks, ranked by complexity:

1. Addition of nodes: users can create new profiles.
2. Addition of edges: users can follow other users.
3. Deletion of nodes and edges: users can delete their accounts or unfollow users they previously followed.
4. Graph rewiring: simultaneous addition and removal of edges and nodes

If the consideration of weights is already implemented, further improvements can be made based on the weight, implying importance, of the edge that had been deleted. When working with deletion of nodes, the graph must again be updated and all edges that this node had must be removed. This is a far more challenging procedure, as several passes must be made in order to ensure the best possible results. The steps should be done as follows: firstly the edges of the deleted node must be removed. Secondly, the graph must be reclustered to accommodate for the changes this leads to. Keeping track of the neighbours of nodes, as is done by the

NCLiC algorithm, may be too computationally heavy in terms of performance. The operations required to accomplish this would be to loop through a node's list of neighbours and delete the node from the neighbour's list, followed by the deletion of the list itself. In terms of asymptotic time this would take $O(n)$ for looping through and accessing the list of n neighbours and $O(m)$ finding and deleting an entry from a list of the chosen node's m neighbours in addition to $O(1)$ for deleting the list itself, leading to $O(nm)$ time complexity. Another approach may thus be needed to speed this up. A possible solution to this may be to use a smart hashing algorithm.

21.1.5 Additional merging algorithms

During the exploration phase of this thesis, many different algorithms have been tested. While Leiden appears to be the best possible algorithm currently available, some algorithms may have been overlooked. Additionally, the work on clustering algorithms is always ongoing and the possibilities of good clustering algorithms are ever expanding. The testing framework created during this thesis is flexible enough to exchange the *base clustering algorithm*, and this is advised in future work. Another unexplored approach is to skip a base clustering algorithm altogether and develop an approach where the clustering is integrated in the merging process. The NCLiC algorithms relies on receiving partitions that can then be modified through the merging process. An integrated clustering could lead to potentially significant performance benefits. However, the challenge with this approach is that the clustering algorithm does not have the information about the whole graph, but only the last partition that it receives. Due to the potential difficulties of implementation, working on this solution has been deemed out of scope.

21.2 Testing Framework

21.2.1 Add other goodness measures

The primary fitness function for evaluating clustering that has been used throughout this thesis is modularity. The reasons for that are listed in Chapter 10.5. While using modularity for comparing different algorithms is easy and gives a good basis for testing, it may be interesting to look at other measures. Two commonly used measures of clustering cohesion are **Silhouette coefficient** and **Dunn index** [83]. The Silhouette analysis measures how well an observation is clustered by estimating the average distance between clusters. The Dunn index also uses distance between nodes to calculate how good the discovered clusters are. Apart from these, several other methods of checking *correctness* of clustering exist. These include the **NMI** (Normalized Mutual Information) score [58], which compares the clusters discovered by the algorithm to the ground truth. By adding new ways to test the performance of clustering algorithms, the framework would become an even better tool for supporting research more effectively.

21.2.2 Increase usability

The testing framework created in this thesis has been worked on by only one person with little focus on usability and readability. While the framework has some help text when running through terminal, it is minimal and may be hard to use for people not familiar with it. It has been integral in the research for this thesis, and could be made more user friendly. In order to make it so, several elements should be implemented. First and foremost, the framework is written in Python and should be translated to a more robust language, like Java or C/C++, removing the speed of the framework as a limitation of testing. Additionally, the framework could be made more modular than it is today, allowing users to pick and choose more easily which algorithms should be tested and which measures should be reported. Doing this would possibly speed up the work of this exciting research topic. However, one should consider if the time and cost of this is warranted, as other matters mentioned above, are deemed to be of higher priority.

Chapter 22

Conclusion

Graph theory is an exceedingly wide and diverse field, but is getting more important as the amount of data available grows. With the onset of social media, this has never been more true than now. The problems with boundless content sharing brings with it an infinite amount of possibilities and just as many problems. As we have seen in recent years, the consequences of actions online can, in many cases, be felt in real life as well. This is why projects like UMOD will continue to gain value and importance.

In this thesis, we performed extensive tests on several of the current state of the art clustering algorithm. We have also outlined and presented a novel way of looking at on-line networks where data is incrementally arriving in partitions, called **incremental graphs**. These types of graphs share similarities with on-line and off-line networks, but have enough differences to warrant a separation between the two types. Working with incremental graphs presents new challenges, and a set of requirements was proposed for working with these types of networks. Based on these requirements a novel algorithm for clustering these types of graphs, called NCLiC, has been proposed. The algorithm is fast and has a high average modularity retention, compared to the baseline set by the Leiden algorithm.

The main challenge of working with incremental graphs is shared with streaming-based graphs, namely that the data should be considered infinite and at this point in time there are no clustering algorithms that work on these types of graphs. The approach to clustering incremental graphs proposed in this thesis is versatile and extensible, making it easy to implement and adapt according to the needs of the user.

The NCLiC algorithm works by merging partitions of networks with a graph stored in memory. By doing this in an efficient manner, the algorithm is able to detect good clusters in a fraction of the time it would take to recluster the graph each time new data arrives. The algorithm is also highly versatile and modular, which makes it easy to improve. The algorithm revolves around the notion that the neighbouring nodes are often a part of the same cluster. This notion is grounded in a trivial observation of the reality of social interaction - the chance of two persons forming a connection when they know each other is higher than if they had never

met. This can be further extrapolated to the notion of *"if many of your friends share an interest, it is probable that you will share that interest as well"*.

NCLiC shows promising results on most types of graphs and in several cases, appears to perform better than some of the older, established and popular clustering algorithms. When compared to a streaming algorithm, SCoDA, NCLiC detects better clusters. NCLiC is also fast, and even the Python implementation promises to save time, when compared to the alternative method of reclustering the whole graph every time new data arrives. During tests it was much faster than the popular Infomap algorithm. With a more performance focused implementation, the algorithm would be fast enough for massive graphs.

Throughout this thesis many different types of approaches has been tested and the results of these tests have been presented. My hope is that the work presented here may be useful for potential future research in the field of community detection and graph theory.

Bibliography

[1] Quote Investigator. *A Lie Can Travel Halfway Around the World While the Truth Is Putting On Its Shoes*. URL: https://quoteinvestigator.com/2014/07/13/truth/.

[2] New York Times: Niraj Chokshi. *That Wasn't Mark Twain: How a Misquotation Is Born*. URL: https://www.nytimes.com/2017/04/26/books/famous-misquotations.html.

[3] Daniel Thilo Schroeder, Konstantin Pogorelov and Johannes Langguth. 'FACT: a Framework for Analysis and Capture of Twitter Graphs'. In: IEEE, 2019, pp. 134–141. DOI: 10.1109/SNAMS.2019.8931870.

[4] Business Insider. *The 10 most-viewed fake-news stories on Facebook in 2019 were just revealed in a new report*. URL: https://www.businessinsider.com/most-viewed-fake-news-stories-shared-on-facebook-2019-2019-11?r=US&IR=T#10-joe-biden-calls-trump-supporters-dregs-of-society-1.

[5] BCC News. *QAnon: What is it and where did it come from?* URL: https://www.bbc.com/news/53498434.

[6] The New York Times. *'PizzaGate' Conspiracy Theory Thrives Anew in the TikTok Era*. URL: https://www.nytimes.com/2020/06/27/technology/pizzagate-justin-bieber-qanon-tiktok.html.

[7] Gabrielle Bruney Michael Sebastian. *Years After Being Debunked, Interest in Pizzagate Is Rising—Again*. URL: https://www.esquire.com/news-politics/news/a51268/what-is-pizzagate/.

[8] Vox. *How the 5G coronavirus conspiracy theory went from fringe to mainstream*. URL: https://www.vox.com/recode/2020/4/24/21231085/coronavirus-5g-conspiracy-theory-covid-facebook-youtube.

[9] The New York Times. *Cambridge Analytica and Facebook: The Scandal and the Fallout So Far*. URL: https://www.nytimes.com/2018/04/04/us/politics/cambridge-analytica-scandal-fallout.html.

[10] New York Post. *Twitter is tweaking misinformation flags to make them easier to spot*. URL: https://nypost.com/2020/10/06/twitter-tweaks-misinformation-flags-to-make-them-easier-to-spot/.

[11] Facebook. *Working to Stop Misinformation and False News*. URL: https://www.facebook.com/formedia/blog/working-to-stop-misinformation-and-false-news.

[12] Simula. *UMOD: Understanding and Monitoring Digital Wildfires*. URL: https://www.simula.no/research/projects/umod-understanding-and-monitoring-digital-wildfires.

[13] Niall J. Conroy, Victoria L. Rubin and Yimin Chen. 'Automatic Deception Detection: Methods for Finding Fake News'. In: *Proceedings of the 78th ASIST Annual Meeting: Information Science with Impact: Research in and for the Community*. ASIST '15. St. Louis, Missouri: American Society for Information Science, 2015. ISBN: 087715547X.

[14] *Computing Veracity – the Fourth Challenge of Big Data*. URL: https://www.pheme.eu/.

[15] Santo Fortunato. 'Community detection in graphs'. In: *Physics Reports* 486.3-5 (Feb. 2010), pp. 75–174. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2009.11.002. URL: http://dx.doi.org/10.1016/j.physrep.2009.11.002.

[16] Albert-Lázsló Barabási. *Network Science*. Cambridge, United Kingdom: Cambridge University Press, 2016.

[17] Brittanica. *Graph Theory*. URL: https://www.britannica.com/topic/graph-theory.

[18] AWS Amazon. *What is Streaming Data?* URL: https://aws.amazon.com/streaming-data/.

[19] Ziv Bar-Yossef, Ravi Kumar and D. Sivakumar. 'Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs'. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 623–632. ISBN: 089871513X.

[20] Michael Elkin and Jian Zhang. 'Efficient algorithms for constructing $(1+\epsilon,\beta)$-spanners in the distributed and streaming models'. In: *Distributed Computing* 18 (Apr. 2006), pp. 375–385. DOI: 10.1007/s00446-005-0147-2.

[21] 'Pearson's Correlation Coefficient'. In: *Encyclopedia of Public Health*. Ed. by Wilhelm Kirch. Dordrecht: Springer Netherlands, 2008, pp. 1090–1091. ISBN: 978-1-4020-5614-7. DOI: 10.1007/978-1-4020-5614-7_2569. URL: https://doi.org/10.1007/978-1-4020-5614-7_2569.

[22] Alex Pothen. *Graph Partitioning Algorithms with Applications to Scientific Computing*. Tech. rep. USA, 1997.

[23] Jierui Xie, Stephen Kelley and Boleslaw K. Szymanski. 'Overlapping Community Detection in Networks: The State-of-the-Art and Comparative Study'. In: 45.4 (Aug. 2013). ISSN: 0360-0300. DOI: 10.1145/2501654.2501657. URL: https://doi.org/10.1145/2501654.2501657.

[24] J. Chitra Devi and E. Poovammal. 'An Analysis of Overlapping Community Detection Algorithms in Social Networks'. In: *Procedia Computer Science* 89 (2016). Twelfth International Conference on Communication Networks, ICCN 2016, August 19– 21, 2016, Bangalore, India Twelfth International Conference on Data Mining and Warehousing, ICDMW 2016, August 19-21, 2016, Bangalore, India Twelfth International Conference on Image and Signal Processing, ICISP 2016, August 19-21, 2016, Bangalore, India, pp. 349–358. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2016.06.082. URL: http://www.sciencedirect.com/science/article/pii/S1877050916311474.

[25] 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.

[26] M. E. J. Newman. In: *PNAS* 103.23 (June 2006), pp. 8577–8582. DOI: https://doi.org/10.1073/pnas.0601602103.

[27] Xin Jin and Jiawei Han. 'Partitional Clustering'. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 766–766. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_631. URL: https://doi.org/10.1007/978-0-387-30164-8_631.

[28] Michal J. Garbade. *Understanding K-means Clustering in Machine Learning*. URL: https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1.

[29] Hamid Bouchachia. 'Dynamic Clustering'. In: *Evolving Systems* 3 (Sept. 2012). DOI: 10.1007/s12530-012-9062-5.

[30] R.L. Winkler. *An Introduction to Bayesian Inference and Decision*. International series in decision processes. Holt, Rinehart and Winston, 1972. ISBN: 9780030813276. URL: https://books.google.no/books?id=GHB2QgAACAAJ.

[31] 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.

[32] Jure Leskovec et al. *Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters*. 2008. arXiv: 0810.1355 [cs.DS].

[33] Pollner, P., Palla, G. and Vicsek, T. 'Preferential attachment of communities: The same principle, but a higher level'. In: *Europhys. Lett.* 73.3 (2006), pp. 478–484. DOI: 10.1209/epl/i2005-10414-6. URL: https://doi.org/10.1209/epl/i2005-10414-6.

[34] Renaud Labiotte Vincent D. Blondel Jean-Loup Guillaume and Etienne Lefebvre. 'Fast unfolding of communities in large networks'. In: *J. Stat. Mech.* (July 2008). DOI: https://doi.org/10.1088/1742-5468/2008/10/P10008.

[35] Joshua R. Tyler, Dennis M. Wilkinson and Bernardo A. Huberman. 'Email as Spectroscopy: Automated Discovery of Community Structure within Organizations'. In: *Communities and Technologies*. NLD: Kluwer, B.V., 2003, pp. 81–96. ISBN: 1402016115.

[36] Amanda L. Traud et al. 'Comparing Community Structure to Characteristics in Online Collegiate Social Networks'. In: *SIAM Review* 53.3 (Jan. 2011), pp. 526–543. ISSN: 1095-7200. DOI: 10.1137/080734315. URL: http://dx.doi.org/10.1137/080734315.

[37] D. Axelsson M. Rosvall. 'The map equation'. In: *The European Physical Journal Special Topics* 178.1 (Nov. 2009), p. 10. DOI: https://doi.org/10.1140/epjst/e2010-01179-1.

[38] D. A. Huffman. 'A Method for the Construction of Minimum-Redundancy Codes'. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[39] Lawrence Page et al. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, Nov. 1999. URL: http://ilpubs.stanford.edu:8090/422/.

[40] L. Waltman V.A. Traag and N.J. van Eck. 'From Louvain to Leiden: guaranteeing well-connected communities'. In: *Sci Rep* 9.5233 (Mar. 2019), p. 12. DOI: https://doi.org/10.1038/s41598-019-41695-z.

[41] Ludo Waltman and Nees Jan van Eck. 'A smart local moving algorithm for large-scale modularity-based community detection'. In: *The European Physical Journal B* 86.11 (Nov. 2013). ISSN: 1434-6036. DOI: 10.1140/epjb/e2013-40829-0. URL: http://dx.doi.org/10.1140/epjb/e2013-40829-0.

[42] Naoto Ozaki, Hiroshi Tezuka and Mary Inaba. 'A Simple Acceleration Method for the Louvain Algorithm'. In: *International Journal of Computer and Electrical Engineering* 8 (Jan. 2016), pp. 207–218. DOI: 10.17706/IJCEE.2016.8.3.207-218.

[43] Seung-Hee Bae et al. 'Scalable and Efficient Flow-Based Community Detection for Large-Scale Graph Analysis'. In: *ACM Trans. Knowl. Discov. Data* 11.3 (Mar. 2017). ISSN: 1556-4681. DOI: 10.1145/2992785. URL: https://doi.org/10.1145/2992785.

[44] V. A. Traag. 'Faster unfolding of communities: Speeding up the Louvain algorithm'. In: *Phys. Rev. E* 92 (3 Sept. 2015), p. 032801. DOI: 10.1103/PhysRevE.92.032801. URL: https://link.aps.org/doi/10.1103/PhysRevE.92.032801.

[45] Alexandre Hollocou et al. 'A linear streaming algorithm for community detection in very large networks'. In: *CoRR* abs/1703.02955 (2017). arXiv: 1703.02955. URL: http://arxiv.org/abs/1703.02955.

[46] Joan Feigenbaum et al. 'On Graph Problems in a Semi-streaming Model'. In: July 2004, pp. 531–543. ISBN: 978-3-540-22849-3. DOI: 10.1007/978-3-540-27836-8_46.

[47] András A. Benczúr and David R. Karger. 'Approximating s-t Minimum Cuts in Õ(n²) Time'. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 47–55. ISBN: 0897917855. DOI: 10.1145/237814.237827. URL: https://doi.org/10.1145/237814.237827.

[48] Arnau Prat-Pérez, David Dominguez-Sal and Josep-Lluis Larriba-Pey. 'High quality, scalable and parallel community detection for large real graphs'. In: Apr. 2014, pp. 225–236. DOI: 10.1145/2566486.2568010.

[49] Learning Hub: Davin Pickell. *Qualitative vs Quantitative Data – What's the Difference?* URL: https://learn.g2.com/qualitative-vs-quantitative-data.

[50] U. Brandes et al. *Maximizing Modularity is hard*. Sept. 2006.

[51] TutorialsPoint. *Data Structures - Asymptotic Analysis*. URL: https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.html.

[52] GeeksforGeeks. *Analysis of Algorithms | Big-O analysis*. URL: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/.

[53] GeeksforGeeks. *Analysis of Algorithms | Set 3 (Asymptotic Notations)*. URL: https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/.

[54] Cmglee. *File: Comparison computational complexity.svg*. URL: https://en.wikipedia.org/wiki/File:Comparison_computational_complexity.svg.

[55] Konect. *Zachary Karate Club*. URL: http://konect.cc/networks/ucidata-zachary/.

[56] Stanford University. *Social circles: Twitter*. URL: https://snap.stanford.edu/data/ego-Twitter.html.

[57] SNAP. *email-Eu-core network*. URL: https://snap.stanford.edu/data/email-Eu-core.html.

[58] Simone Romano et al. 'Standardized Mutual Information for Clustering Comparisons: One Step Further in Adjustment for Chance'. In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, June 2014, pp. 1143–1151. URL: http://proceedings.mlr.press/v32/romano14.html.

[59] Richard M. Karp. 'On-Line Algorithms Versus Off-Line Algorithms: How Much is It Worth to Know the Future?' In: *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1 - Volume I*. NLD: North-Holland Publishing Co., 1992, pp. 416–429. ISBN: 044489747X.

[60] Wayne W Zachary. 'An information flow model for conflict and fission in small groups'. In: *Journal of anthropological research* (1977), pp. 452–473.

[61] NetworkX. *Software for Complex Networks*. URL: https://networkx.org/documentation/stable/index.html.

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

[63] Timothy A. Davis and Yifan Hu. 'The University of Florida Sparse Matrix Collection'. In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: https://doi.org/10.1145/2049662.2049663.

[64] SuiteSparce Matrix Collection. *Group DIMACS10*. URL: https://sparse.tamu.edu/DIMACS10.

[65] V. Batagelj and M. Zaversnik. *An O(m) Algorithm for Cores Decomposition of Networks*. 2003. arXiv: cs/0310049 [cs.DS].

[66] Illés Farkas Gergely Palla Imre Derényi and Tamás Vicsek. 'Uncovering the overlapping community structure of complex networks in nature and society'. In: *Nature* 10 (2005), pp. 814–818. DOI: 10.1038/nature03607.

[67] Shie Mannor et al. 'K-Means Clustering'. In: *Encyclopedia of Machine Learning*. Springer US, 2011, pp. 563–564. DOI: 10.1007/978-0-387-30164-8_425. URL: https://doi.org/10.1007%5C%2F978-0-387-30164-8_425.

[68] Aaron Clauset, M. E. J. Newman and Cristopher Moore. 'Finding community structure in very large networks'. In: *Physical Review E* 70.6 (Dec. 2004). ISSN: 1550-2376. DOI: 10.1103/physreve.70.066111. URL: http://dx.doi.org/10.1103/PhysRevE.70.066111.

[69] Gephi. *The Open Graph Viz Platform*. URL: https://gephi.org/.

[70] JetBrains. *The Python IDE for Professional Developers*. URL: https://www.jetbrains.com/pycharm/.

[71] NumPy. *The fundamental package for scientific computing with Python*. URL: https://numpy.org/.

[72] Python Igraph. URL: https://igraph.org/python/.

[73] NetworKit. *Large-scale Network Analysis*. URL: http://wiki.ex3.simula.no/doku.php.

[74] Simula Ex3. *Simula Research Laboratory AS research cluster*. URL: https://www.jetbrains.com/pycharm/.

[75] Oracle Labs. *Infomap*. URL: https://docs.oracle.com/cd/E56133_01/2.6.1/reference/algorithms/weighted_infomap.html.

[76] V Nicosia et al. 'Extending the definition of modularity to directed graphs with overlapping communities'. In: *Journal of Statistical Mechanics: Theory and Experiment* 2009.03 (Mar. 2009), P03024. DOI: 10.1088/1742-5468/2009/03/p03024. URL: https://doi.org/10.1088%5C%2F1742-5468%5C%2F2009%5C%2F03%5C%2Fp03024.

[77] Nicolas Dugué and Anthony Perez. *Directed Louvain : maximizing modularity in directed networks*. Nov. 2015. DOI: 10.13140/RG.2.1.4497.0328.

[78] Alexandre Hollocou. *GitHub repository: SCODA: A linear streaming algorithm for community detection in very large networks*. URL: https://github.com/ahollocou/scoda.

[79] NetworkX. *networkx.algorithms.assortativity.average_neighbor_degree*. URL: https://networkx.org/documentation/stable//reference/algorithms/generated/networkx.algorithms.assortativity.average_neighbor_degree.html.

[80] Stanford Network Analysis Project. *email-Eu-core network*. URL: https://snap.stanford.edu/data/email-Eu-core.html.

[81] Stanford Network Analysis Project. *Youtube social network*. URL: https://snap.stanford.edu/data/com-Youtube.html.

[82] Stanford Network Analysis Project. *Amazon product co-purchasing network*. URL: https://snap.stanford.edu/data/com-Amazon.html.

[83] Data Novia. *Cluster Validation Statistics: Must Know Methods*. URL: https://www.datanovia.com/en/lessons/cluster-validation-statistics-must-know-methods/.