Scalable Community Detection

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

Dissertation

von

Michael Alexander Hamann

Tag der mündlichen Prüfung:	23. Juli 2020
1. Referentin:	Prof. Dr. Dorothea Wagner
2. Referent:	Prof. Dr. Ulrik Brandes

Acknowledgements

First of all, I would like to thank Dorothea Wagner for offering me the opportunity to work as a doctoral researcher in her group. While leaving me a lot of freedom to pursue the research projects and collaborations of my choice she also ensured that I stayed focused on finishing this thesis. Many thanks also to Ulrik Brandes for agreeing to be the second referee of my thesis. Further, I would like to thank all my colleagues, all the students I supervised and all my co-authors for the many fruitful and interesting discussions we had. Many of the results in this thesis would not exist without these discussions. In particular, I would like to thank the colleagues with whom I shared my office, first Ben Strasser and later Lars Gottesbüren, with whom I could always discuss my research ideas and who provided many ideas, feedback and guidance.

I would also like to thank my family. My parents for encouraging me to follow my interests and supporting my wish to study informatics. My wife for always supporting me even when I worked long hours or traveled to attend conferences and summer schools, in particular with caring for our daughter such that I could concentrate on my work for this thesis. Last but not least I would also like to thank my daughter – she is such a lovely and happy girl, it is always a joy to be with her, in particular in these difficult times of the coronavirus pandemic.

My work on this thesis and all results presented in it has been funded by the German Research Foundation (DFG) under grant WA 654/22-1 and WA 654/22-2 within the Priority Programme 1736 "Algorithms for Big Data". For the experiments on distributed clustering (Chapter 6) and exact editing (Chapter 10), we further acknowledge support by the state of Baden-Württemberg through bwHPC.

Contents

1 Introduction

A graph consists of objects, called *nodes*, that are connected by *edges*. These connections may have a certain strength or a direction. Graphs can be used in a lot of contexts where objects have relations to each other [Fon+11]. A common example throughout this thesis are social networks, where the nodes represent humans that are, e.g., connected by friendships. Other examples are scientists as nodes that are connected by an edge if they co-authored a paper, or the World Wide Web, where nodes are websites and edges represent links between them. Graphs are also used in biology, e.g., to model proteins as nodes and interactions between them as edges.

Given a large graph, we might wonder how it is structured. While for small graphs, a visualization might give important insights, this becomes increasingly difficult with the size of the graph. Frequently, graphs do not have edges distributed uniformly at random, but there are groups of nodes that are internally densely connected with few connections to other nodes. Such groups are commonly called *communities* or *clusters*. Detecting communities, a common task in network analysis, helps us understand the overall structure of graphs but also provides us with sets of nodes we should look at together. In a protein interaction network, a community might be a group of proteins with similar function while in a social network it might be a group of friends.

Community detection is a well researched topic [Sch07; For10; AP14; Har+14; FH16; HKW16; Cha+17; Cop+19]. While there is no universally accepted definition of a good community, it is commonly accepted that communities should be internally densely and externally sparsely connected. There have been a lot of measures proposed to formalize this fuzzy concept [Cha+17]. We consider two formalizations for disjoint communities following this principle: the famous modularity [NG04] and the more recently proposed map equation [RAB09]. As modularity optimization is NP-hard [Bra+08], heuristics such as the well-known Louvain algorithm [Blo+08] are used in practice. Similarly, also for the map equation we are only aware of heuristics [RAB09; BH15; Bae+17; ZY18; Fun19; FA19]. Most of these heuristics are based on the idea of moving nodes between communities to improve the quality measure, combined with hierarchical contractions that combine several nodes into a single node and thus allow moving nodes together.

A different concept is to consider an ideal model for communities like a disjoint union of complete graphs, also called cliques. We are then looking for such a graph that is close to a given graph in the sense that we only need to remove and insert few edges. This is called cluster editing [BB13]. In biology, such models are used to detect communities for example in protein interaction graphs where the strength of interaction between proteins gives natural costs for edge insertions and deletions [Böc+09; Wit+11].

All of these approaches have in common that they partition a graph into disjoint communities such that every node belongs to exactly one community. While this model is quite well-understood, it cannot model real-world scenarios where nodes belong to more than a single community. For example in a social network, a person might be part of a group of friends at work, a group of friends from school, a group of friends at a sport club and also a close family circle [ABL10; ML12]. Therefore, a lot of models and algorithms for overlapping communities have been proposed. For overlapping communities, there is a much stronger focus on the quality of individual communities as it is more challenging to come up with good global quality measures that can be optimized efficiently [XKS13; AP14; FH16]. As these scenarios are a lot more complex, and we cannot expect communities to be clearly separated, such algorithms usually work only in very specific conditions or need a lot of running time [Buz+14; Gel19].

Many real-world networks evolve over time, for example friendships in a social network change over time, new users join a social network and others leave it. Further, we can also define graphs based on temporal features like a social network where an edge exists only if two people have interacted in a certain time frame. When detecting communities on such graphs, a natural request is that we do not want communities to change dramatically from one time step to another unless there was a dramatic change in the network structure [HKW16; CR19]. This means that community detection algorithms for evolving networks should take the community structure of previous time steps into account – not just to save running time, but also to produce better results.

An additional challenge is the ever-increasing amount of data that is available today. This makes it necessary to consider scenarios where the whole graph does not fit into the RAM of a single computer anymore. Scaling the computation across a cluster of compute servers is a natural way to handle such massive graphs. Another one is to use so-called external memory, nowadays typically SSDs, to store data and to use access patterns that suit them. Further, we might also simply discard parts of the graph [SPR11] and hope we still have enough left to accurately detect communities. We might also not be interested in communities of the whole graph but only communities containing a certain node or a group of nodes. Here, we can use local algorithms that avoid visiting the whole graph.

An important part of the design of a community detection algorithm is to evaluate how well it actually works. Due to the lack of a commonly agreed on definition what constitutes a good community, this is a difficult problem in itself. In most graphs derived from real-world networks, we either do not know what communities we are looking for or if they can even be detected [Bad+14; FH16]. As a remedy, synthetic benchmark graph generators have been designed. They provide a graph and a community structure that we can expect to be detectable due to the way the graph is generated. At the same time, such benchmark graphs try to construct graphs that have properties such as a skewed degree distribution typically found in real-world networks. LFR benchmark graphs [LFR08; LF09a] are the most widely used benchmark graphs for community detection [FH16; Emm+16]. They support disjoint and overlapping communities as well as optionally weighted and/or directed graphs. The more recently proposed CKB benchmark [Chy+14] considers only overlapping communities, with a skewed distribution of the number of communities per node.

1.1 Our Contribution

The first part of this thesis deals with benchmarking community detection algorithms. We give an overview of existing approaches and provide two new graph generators. The remaining parts then deal with different models of communities – disjoint density-based communities, editing, and local communities. For each of these models, we introduce new algorithms that improve the state-of-the-art in terms of quality and/or in terms of scalability.

The first part starts with an overview of existing benchmark graph generators for community detection as well as networks that are frequently used as benchmark instances. We then present two novel graph generators. In Chapter 3, we present the first external memory graph generator that implements the LFR benchmark graph model and is able to generate graphs larger than the RAM of a single compute server. It outperforms the original implementation of the LFR benchmark already for graphs still fitting into RAM while producing graphs that closely follow the original model. In Chapter 4, we present our second graph generator that deals with dynamically changing communities in a dynamic graph model based on the CKB model. We show that our generator is able to create graphs that have stable properties over time while featuring a significantly changing community structure with smooth changes.

In the second part, we explore two approaches to make existing disjoint community detection algorithms scalable. In Chapter 6, we present a distributed community detection algorithm that can optimize modularity and map equation. It is based on the Thrill framework [Bin+16] and the idea of moving several nodes at once between communities. We show that it outperforms the state-of-the-art distributed algorithm for optimizing map equation in terms of the quality of the found community structure while being faster and needing less memory. Chapter 7 explores how removing edges according to different rankings affects the performance of clustering algorithms. We show that while there are methods that very effectively select intra-cluster edges in LFR benchmark graphs, they lead to vastly different communities on real-world graphs. On the other hand, random sampling seems to be a promising technique that, up to certain rates, keeps the community structure mostly intact.

In the third part, we propose scalable algorithms for the problem of editing a given graph to a quasi-threshold graph, a model proposed for detecting communities in social networks [NG13]. In Chapter 9, we present a scalable algorithm that optimizes this problem heuristically. We show that our algorithm scales well to graphs with hundreds of millions of edges. In order to examine its quality, in Chapter 10, we also consider solving the problem exactly using a combinatorial branch and bound algorithm and an integer linear program – albeit on much smaller graphs. Our branch and bound algorithm is also able to efficiently enumerate all solutions, allowing an analysis of all found solutions – for some graphs, there are thousands that differ significantly. We show that our heuristic algorithm is frequently exact or close to a best solution. Further, our branch and bound algorithm is able to exploit parallelism more efficiently than the commercial Gurobi ILP solver, making it faster than the ILP even though it enumerates all solutions.

The last part deals with detecting local communities around a given seed node without exploring the whole graph. In Chapter 11, we conduct an extensive experimental study and show that starting from the largest clique in the neighborhood of the seed node significantly improves results on synthetic as well as real-world networks.

For all our algorithms, we also publish implementations. Links to them are provided in the respective chapters. We implemented many of our algorithms in *NetworKit* [SSM16]. Some of them are already included in NetworKit, for others we provide extended versions of NetworKit that we intend to contribute to NetworKit for future releases. NetworKit combines efficient graph algorithms implemented in C++ with a Python interface. This allows the interactive analysis of networks and makes it easy to write scripts that implement experimental workflows. We use this to analyze also the results of algorithms that were not implemented in NetworKit like our external memory LFR generator. For some of these evaluations, we also contributed new algorithms to NetworKit, e.g., to analyze the difference between two graphs for our work on exact editing (Chapter 10).

The remainder of this introduction introduces the notation and important concepts used in the remainder of the thesis.

1.2 Preliminaries

In this section we introduce the notation as well as important concepts that we use throughout this thesis. The notation is mostly the same as the one we used in the papers the individual chapters are based on. This section is based on the corresponding section in our papers on external memory graph generation [Ham+18b] and on local community detection [HRW17]. The former is joint work with Ulrich Meyer, Manuel Penschuck, Hung Tran and Dorothea Wagner, while the latter is joint work with Eike Röhrs and Dorothea Wagner.

We define $[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_{>0}$. The notation $[x_i]_{i=a}^{b}$ denotes an ordered sequence $[x_a, x_{a+1}, \dots, x_b]$.

Graphs. A graph G is a tuple $G = (V, E)$ of $n := |V|$ sequentially numbered nodes $V = \{v_1, \dots, v_n\}$ and $m := |E|$ edges. Unless stated differently, we assume that graphs are undirected. To obtain a unique representation of an undirected edge $\{u, v\} \in E$ as a tuple, we sometimes use ordered edges $[v_i, v_j] \in E$ implying $i \leq j$. The edge is still undirected, the ordering is just used algorithmically but does not carry any meaning. An edge $\{u, v\}$ is *incident* to a node v_i, if $v_i \in \{u, v\}$. Two nodes v_i, v_j are *adjacent* if they are connected by an edge, i.e., $\{v_i, v_j\} \in E$. In general, we assume that our input graphs are *simple*, i.e., without self-loops and multi-edges. In some algorithms though, we deal with graphs with self-loops and multi-edges in intermediate steps.

In some cases, we also consider *weighted graphs* $G = (V, E, w)$ with *weights* $w : E \to \mathbb{R}_{>0}$. We then treat unweighted graphs as weighted graphs with $w(v_i, v_j) = 1$ for every edge $\{v_i, v_j\} \in E$.

The *neighbors* $N(v_i) = \{v_j | \{v_i, v_j\} \in E\}$ of a node v_i are the nodes that are adjacent to it. For a set of nodes $S \subseteq V$, we denote by $N(S)$ the union of all neighborhoods of the nodes in S without S, i.e., $N(S) := \cup_{u \in S} N(u) \setminus S$.

The *degree* $\deg(v_i) := |N(v_i)|$ of a node v_i is its number of neighbors. Let Δ denote the maximum degree over all nodes in V. For weighted graphs, the *weighted degree* $\deg_w(v_i) = \sum_{v_j \in N(_vi)} w(\{v_i, v_j\})$ is the sum of the edge weights of all edges incident to u. For (weighted) degrees, we count self-loops twice. $\mathcal{D} = [\, d_i \,]_{i=1}^n$ is a degree sequence of graph G iff $\forall v_i \in V\colon \deg(v_i) = d_i$.

Communities. A *community* or *cluster*[1] $C \subseteq V$ is a node subset of size $n_C := |C|$. A *clustering* $\mathcal{C}$ is a set of clusters. A clustering is *disjoint* if each node $v_i \in V$ is part of exactly one cluster. A disjoint clustering is sometimes also called a *partition* of the nodes while a possibly overlapping clustering is frequently called *cover*. The *shell* $S(C)$ is a synonym for the neighbors of the community C, i.e., $S(C) = N(C)$. The *boundary* $B(C)$ of a community C are the nodes in C that have neighbors in the shell $S(C)$, i.e., $B(C) = \{v_i \in C \mid N(v_i) \cap S(C) \neq \emptyset\}$.

Cut, Volume. The *volume* $\mathrm{vol}(S)$ of a node set S is the sum of their degrees. The *weighted volume* $\mathrm{vol}_w(S)$ of a node set S is analogously the sum of their weighted degrees. We define the *weighted cut* of two disjoint node sets $S, T \subseteq V$, $S \cap T = \emptyset$ as $\mathrm{cut}_w(S,T) := \sum_{u \in S, v \in T} w(\{u,v\})$. As a simplification, we write $\mathrm{cut}_w(v,S) := \mathrm{cut}_w(\{v\},S)$ for the weighted cut between a node and a node set S and $\mathrm{cut}_w(S) := \mathrm{cut}_w(S, V \setminus S)$ for the cut between S and the rest of the graph.

Clique. A *clique* is a set of nodes that induce a complete subgraph in G. In other words, $A \subseteq V$ is a clique, if and only if $\forall u \in A, \forall v \in A \setminus \{u\} : \{u,v\} \in E$. Then, a *maximal* clique Q is a clique that can not be enlarged by adding a node. A clique is called a *triangle* if it consists of exactly three nodes.

Randomization and Distributions. $\textsc{Pld}\,([a,b),\gamma)$ denotes an integer Powerlaw Distribution with exponent $-\gamma \in \mathbb{R}$ for $\gamma \geq 1$ and values from the interval $[a,b)$; let X be an integer random variable drawn from $\textsc{Pld}\,([a,b),\gamma)$ then $\mathbb{P}[X{=}k] \propto k^{-\gamma}$ (proportional to) if $a \leq k < b$ and $\mathbb{P}[X{=}k] = 0$ otherwise. For $X = [\, x_i \,]_{i=1}^n$, we define the *mean* $\langle X \rangle := \sum_{i=1}^n x_i/n$ and the *second moment* $\langle X^2 \rangle := \sum_{i=1}^n x_i^2/n$ of the sequence X. A statement depending on some $x > 0$ is said to hold *with high probability* if it is satisfied with probability at least $1 - 1/x^c$ for some constant $c \geq 1$.

1.3 Common Measures

In this section, we introduce common measures for properties of graphs and in particular communities. We primarily mention those that are used in several parts of this thesis, but also some additional measures in Section 1.3.5.

[1]In this thesis, we use both terms interchangeably.

1.3.1 Average Local Clustering Coefficient

The *local clustering coefficient* of a node is the density of its neighbors, i.e., how many of its neighbors are connected by an edge divided by the possible number of edges [Kai08; New10]. This just considers the graph and not a particular community structure. The *average local clustering coefficient* is the average of this score among all nodes of the graph. Every edge that connects two neighbors of a node spans a triangle, thus algorithms for listing triangles can be used to compute the clustering coefficient efficiently [OB14]. If the average local clustering coefficient is higher than we expect in a random graph with the same amount of edges, we can assume that edges are not uniformly distributed but that there are denser groups. While this is not a measure of a community structure, it can certainly indicate that we can expect communities. A sufficiently large community can also be internally dense without having a high local clustering coefficient, thus the inverse is not necessarily true. In many graphs from applications like social networks, we expect that communities have many triangles and thus there should be no communities without triangles. In Chapter 11, we show how measures based on very similar ideas can be used to detect communities around a seed.

1.3.2 Conductance

The *conductance* is a measure that formalizes the fuzzy concept of an internally densely and externally sparsely connected community. It is defined as the cut divided by the volume of the community or the remainder of the graph:

$$\textsc{conductance}(C) = \frac{\mathrm{cut}_w(C)}{\min(\mathrm{vol}_w(C), \mathrm{vol}_w(V \setminus C))}.$$

A low conductance thus means that a community is well-separated, i.e., only few of its edges leave it. This does not necessarily mean that the community is well-connected internally. For example, consider a graph consisting of several connected components. While every connected component has a conductance of zero, groups of connected components have a conductance of zero, too. Further, the community consisting of the whole graph always has a conductance of zero. As the conductance measures only the quality of a single community, measures like the average conductance of all communities have been proposed to measure the quality of a whole clustering [GKW14].

1.3.3 Modularity

A simple measure for a clustering is the *coverage*, i.e., the fraction of edges that are inside a community. On its own, this measure is not useful as an optimization criterion, as the coverage is maximal when all edges are inside a single community consisting all nodes of the graph. Newman and Girvan propose to subtract the expected coverage in a random graph with the same node degrees from the actual coverage [NG04]. This measure is called *modularity* and the basis of many community detection algorithms. Using our terminology, it can be defined as follows:

$$\mathcal{Q}(\mathcal{C}) := \sum_{C\in\mathcal{C}} \frac{\mathrm{vol}_w(C) - \mathrm{cut}_w(C)}{\mathrm{vol}_w(V)} - \sum_{C\in\mathcal{C}} \frac{\mathrm{vol}_w(C)^2}{\mathrm{vol}_w(V)^2}$$

Modularity suffers from a so-called *resolution limit* [FB07]. This means that in some cases, modularity scores can be improved by merging small but clearly distinct communities. The size of the merged communities depends on the size of the graph, i.e., on smaller graphs, smaller communities will be found. An effect of this is that community detection algorithms optimizing modularity tend to detect communities that are of relatively uniform size. A resolution parameter has been introduced that allows to control the approximate size of the detected communities [RB06]. While this allows finding smaller (or larger) communities, both cannot be found at the same time [Kum+07].

Finding a clustering with maximum modularity is NP-hard [Bra+08]. Algorithms used in practice are thus heuristics such as the famous Louvain algorithm [Blo+08] that we describe in Section 1.4.1. In Chapter 6, we present a distributed algorithm for optimizing modularity.

1.3.4 Map Equation

The *map equation* [RAB09] is the quality measure optimized by the Infomap algorithm [RAB09]. It is based on the idea that a random walk on a graph, that chooses the next node uniformly at random among the neighbors of the current node, tends to stay in communities. Instead of actually simulating random walks, the map equation measures the expected code word length of the description of such a random walk using a two-level code. There is a code word for each community and there are separate dictionaries of words for each community that contain a code word for every node inside the community as well as an exit code word. The description thus starts with a community code word, then continues with code words of that community until the exit code word is encountered, which is followed by another community code word and so on. A good community structure should lead to shorter code word lengths as random walks are more likely to stay inside communities. Thus, lower scores of the map equation indicate better clusterings. While we use this two-level formulation for undirected graphs, the map equation allows for very natural extensions to hierarchical communities using multi-level codes and also for directed graphs. The expected code word length for two-level codes on undirected graphs can be expressed in the following formula. To simplify its definition, let $\mathrm{plogp}(x) := x \log x$.

$$\begin{aligned} L(\mathcal{C}) :=& \,\mathrm{plogp}\left(\sum_{C\in\mathcal{C}} \frac{\mathrm{cut}_w(C)}{\mathrm{vol}_w(V)}\right) - 2\sum_{C\in\mathcal{C}} \mathrm{plogp}\left(\frac{\mathrm{cut}_w(C)}{\mathrm{vol}_w(V)}\right) \\ &+ \sum_{C\in\mathcal{C}} \mathrm{plogp}\left(\frac{\mathrm{cut}_w(C) + \mathrm{vol}_w(C)}{\mathrm{vol}_w(V)}\right) - \sum_{v\in V} \mathrm{plogp}\left(\frac{\deg_w(v)}{\mathrm{vol}_w(V)}\right) \end{aligned}$$

The last term of the map equation is independent of the actual clustering and can thus be omitted for optimization. The map equation has a resolution limit, too [KR15]. However, in practice it is much less pronounced than for modularity.

We are not aware of any results regarding the complexity of the optimization of the map equation. The heuristic Infomap algorithm [RAB09] has been proposed for optimizing the map equation, it is similar to the Louvain algorithm described in Section 1.4.1. In Chapter 6, we also present a distributed algorithm for optimizing the map equation.

1.3.5 Additional Quality Measures

Surprise [AMM05; AM11] measures how surprising, i.e., unlikely, it is that a random graph has at least as many intra-cluster edges as observed. Finding a clustering with optimum surprise is NP-hard [FKW14]. To allow for fast heuristic algorithms, an approximation has been suggested [TAD15]. While good results on synthetic and real-world graphs have been reported [AM11; TAD15], a recent study [Xia+18] suggests that there might be problems with splitting communities with heterogeneous degrees and detecting too many communities. In a master thesis I supervised [Wie19], we also observed much smaller communities compared to communities found by optimizing map equation or modularity.

Reichardt and Bornholdt [RB06] not only generalize modularity to include a resolution parameter, but also suggest using a simple random Erdős-Rényi graph [Gil59; ER59] as null model.

The *constant Potts model (CPM)* [TDN11] is similar to such generalizations of modularity but does not have a resolution limit. Instead, it has an explicit parameter that determines the resolution of the found communities independently of the provided graph. When choosing this parameter correctly, its performance on synthetic benchmarks is excellent [TDN11].

Significance has been introduced [TKD13] as a way of determining this resolution parameter but can also be used as a quality measure on its own. It is based on a similar idea as Surprise, but instead of considering the probability of a fixed partition containing that many edges, it considers the probability of finding a partition with that many edges in a random graph.

Generative models consider the likelihood of the given graph considering a certain model of communities, in particular stochastic block models. Each block represents a community and the probability of an edge $\{u, v\}$ depends on the blocks of u and v, i.e., for each pair of blocks there is an edge probability. While early models [SN97] do not consider the degrees of the nodes, models with a correction for node degrees show much better results [KN11]. Also overlapping models have been proposed [Pei15] that even allow the comparison of the fit of different models by comparing the minimum description length.

1.4 Community Detection Algorithms

Already a survey in 2010 [For10] listed over 450 references, and since then many more algorithms have been published. Therefore, it is impossible to give a comprehensive overview of the field as part of this thesis. Instead, we highlight a few algorithms that have been shown to perform well in benchmarks and introduce their algorithmic principles in the following sections. In the individual chapters of this thesis, we introduce related work specifically for the different scenarios, i.e., for distributed algorithms in Chapter 6, for editing-based algorithms in Chapter 8 and for local community detection algorithms in Chapter 11.

1.4.1 Louvain

One of the most popular community detection algorithms is the so-called *Louvain* algorithm [Blo+08], named after the university where its authors were back then. It is based on two main components: local moving and contraction, which are applied alternately. The algorithm starts by assigning each node to its own community. Then, it iterates over the nodes of the graph in a random order and for each node u, it checks for each of its neighbors v how much the modularity would increase or decrease when moving u into v's community. If an improvement is possible, u is moved into the neighbor's community that gives the maximum improvement. This process is repeated in rounds until in a round no node has been moved or a stopping criterion is met. Typical stopping criteria include limiting the number of rounds or requiring a minimum modularity improvement. After this local moving phase, the graph is contracted such that each community becomes a node in the contracted graph. To ensure that the modularity is the same after the contraction as before the contraction, edges inside a community are represented as a self-loop. The weight of the self-loop is the sum of the edge weights in the contracted subgraph. Resulting multi-edges between communities are reduced to single edges by summing up their weights. On this contracted graph, the local moving phase is repeated. Local moving and contraction phase are repeated until after a local moving phase each node is still in its own community. In a final uncontraction step, the community assignments of each contracted node is prolonged to the original nodes.

Many variations of this basic algorithm have been proposed. A straightforward extension is refinement [RN11]. For refinement, local moving is executed again to improve the community structure after every uncontraction level. The *Leiden* algorithm [TWE19] additionally more carefully selects which nodes shall be contracted together and uses a strategy where only certain nodes are active to accelerate local moving.

The Louvain algorithm has been adapted for shared-memory parallel as well as distributed scenarios [Que+15; Gho+18a]. NetworKit contains a parallel version of the Louvain method called PLM, optionally with additional refinement (PLMR) [SM16]. The implementation in NetworKit also allows disabling the parallelism and is then equivalent to the original Louvain algorithm.

The *Infomap* algorithm that has been proposed for optimizing the map equation [RAB09] is also based on the Louvain algorithm with additional steps for identifying sub-clusters before contraction and additional refinement steps.

1.4.2 Label Propagation

The basic idea of label propagation is similar to local moving but instead of optimizing a particular measure, each node simply joins the community most of its neighbors are in. Initially, each node is in its own community. Ties are broken uniformly at random. Thus, the first steps are purely random until the first communities start to form. Label propagation is simple and fast but the quality of the found communities depends a lot on the properties of the graph. In particular in dense graph, it can easily happen that a single community emerges that contains all or most nodes. To compensate for this problem, the growth of a community can be dampened by reducing its weight every time it is adopted by a new node or by setting the number of neighbors in relation to the community's size [ŠB11]. Label propagation has also been adapted for overlapping communities by keeping several communities per node. SLPA [XSL11] is such an adaptation. While some benchmarks [XKS13] show that it performs well, other comparisons [Buz+14] cannot confirm this – the most likely explanation is the different parametrization of the used benchmarks. The distributed EgoLP algorithm is another example for overlapping label propagation [Buz+14]. It has 14 parameters, the authors propose two different parameter sets for synthetic benchmarks with three and six overlapping communities per node, again confirming the high sensitivity of label propagation to graph and parameter choices. Implementations that use external memory exist, too [ASS15].

1.4.3 Greedy Clique Expansion (GCE)

Greedy Clique Expansion (GCE) [Lee+10] starts by detecting all maximal cliques of the graph. Each of them is used as the seed of a community that is greedily expanded by adding nodes that minimize a measure similar to the conductance of the community. The authors find that there are both many cliques that have large overlaps and expanded communities that are the same or very similar. Thus, a major part of GCE is filtering such duplicates.

1.4.4 Order Statistics Local Optimization Method (OSLOM)

OSLOM, Order Statistics Local Optimization Method, is an overlapping community detection algorithm [Lan+11]. It uses a measure of statistical significance to evaluate single communities. For a community C, it measures if a prefix of a list of candidates has significantly more connections to C than expected by chance in a random null model. Nodes already part of C are also repeatedly re-evaluated according to this criterion. For each community, OSLOM both considers adding neighboring nodes and removing existing nodes in a cleaning-up procedure. A community is significant if it is non-empty more than half of the times the cleaning-up procedure is applied. OSLOM detects communities by repeatedly expanding single nodes into communities. The procedure stops, if it detects communities similar to existing communities again and again. OSLOM also checks if communities contain significant sub-communities or if merging multiple communities gives significant communities. Thus, OSLOM is also able to detect hierarchical communities. The authors further propose extensions to weighted and directed graphs. Finally, OSLOM can use an existing community structure as starting point of its search, thus allowing the algorithm to be used for dynamic graphs.

1.4.5 MOSES

In contrast to the previously discussed overlapping community detection algorithms, MOSES [MH10] defines a quality measure for overlapping communities that is then optimized. This quality measure is based on a stochastic block model for overlapping communities, the algorithm optimizes the likelihood of the graph given that model. MOSES uses a combination of expanding single edges into communities, moving nodes between communities and deleting single communities. All of these steps greedily optimize the quality measure.

Part I

Benchmark Graphs

2 Introduction to Benchmarking Community Detection Algorithms

When developing or choosing a community detection algorithm, there are usually two questions: how fast is it, and how good are its results. Measuring running times in a reproducible way is not trivial as they might depend on a lot of factors, but given a certain compute system and input instance, it is quite straightforward. On the other hand, measuring the quality of the results is much less straightforward, even given a certain input graph. Depending on the use case, one might want to have smaller or larger, or more or less uniform community sizes, to name just a few parameters. Frequently, graphs from applications are also not available due to data protection or corporate interests. Graphs from applications where we know what communities we want to find are even rarer, and even for those that are frequently used, it is often not clear if those communities can actually be discovered from the graph structure alone [PLC17].

To make comparing community detection algorithms easier and less arbitrary (due to the choice of input graph), synthetic benchmarks for community detection have been introduced [GN02; LFR08]. Due to the structure of the benchmark graph, we expect that good community detection algorithms will be able to detect a certain known ground truth community structure. Over the time, several benchmarks have been proposed, in the following we will introduce some of them. The most popular one of them is certainly the LFR benchmark [LFR08; LF09a], which we also mainly use in this thesis and for which we propose a scalable external memory implementation in Chapter 3. By choosing parameters accordingly, benchmark graphs can for example show that certain community detection algorithms have problems detecting small and large communities at the same time.

Unfortunately, synthetic benchmark graphs also have their own set of problems. This starts with the problem of parameter choice. If, for example, a node has only one edge to nodes of its own community, we cannot reasonably expect community detection algorithms to recover this community. Nevertheless, such configurations of the LFR benchmark with an average degree of ten and nodes that are part of up to eight communities are used to conclude that "all of the algorithms do not yet achieve satisfying performance" for those graphs where half of the nodes are part of more than two communities [XKS13]. Apart from the obvious problem of choosing suitable parameters, the realism of benchmark graphs can also be questioned. Many models use graphs that are completely random apart from the modeled communities, thus featuring a very regular structure. As we will also see in this thesis in Chapter 6 and 7 on distributed clustering and sparsification, many algorithms behave completely differently on real-world and synthetic benchmark graphs. Therefore, while synthetic benchmark graphs provide a good way to examine the quality of a community detection algorithm, they cannot replace an evaluation on graphs from an actual application. It might even be that certain community detection algorithms work only on benchmark graphs or only on certain application graphs as they exploit specific properties of these graphs. For this reason, we also include graphs from applications into our evaluations.

Evaluating detected communities using ground truth communities requires comparing them. Given two sets of communities, we want to know how well they match, ideally as a simple score in the range $[0, 1]$, where 1 means they are identical. Even for disjoint communities, it is not clear when the score should reach zero and how to penalize e.g. merging two communities compared to moving single nodes between communities. For overlapping communities, there are even more degrees of freedom as community detection algorithms can also detect additional communities not present in the ground truth, up to the point of detecting every possible set of nodes as community, which includes all ground truth communities but is certainly not a useful community detection algorithm [PSZ10; FH16; Jeb+18].

In the following sections, we introduce synthetic benchmark graph generators as well as a set of graphs derived from applications that are used throughout this thesis. Further, we discuss measures for comparing communities.

2.1 Synthetic Benchmark Graphs

One of the simplest graph models is Gilbert's $G(n, p)$-model of a graph with n nodes where every edge exists with probability p [Gil59]. The $G(n, m)$ model of a random graph with n nodes and m uniformly distributed edges is closely related [ER59] as $G(n, m)$ is equivalent to $G(n, p)$ when drawing m from a binomial distribution with probability p and $\binom{n}{2}$ trials for unweighted, simple graphs (for graphs with self-loops or directed graphs, the number of trials needs to be adjusted). Both models are commonly known as Erdős-Rényi graphs. While Erdős-Rényi graphs themselves are not particularly interesting as benchmark graphs for community detection, apart from serving as baseline for graphs where no communities should be found, they are the building block of the *stochastic block model*. Its simplest form is the planted partition model, where nodes are split into equally-sized groups. Two nodes inside the same group are connected with probability p while two nodes that belong to different groups are connected with some probability $r < p$ [CK01]. This can be generalized to an arbitrary assignment of nodes to blocks and a matrix of connection probabilities for each combination of two blocks [SN97]. A popular version of this planted partition model is the benchmark proposed by Girvan and Newman [GN02]. It consists of 128 nodes with expected degree 16 that are divided into four groups of 32 nodes each. As graphs with non-uniform community sizes and node degrees are much more challenging for community detection than such uniform communities, the LFR benchmark has been proposed to provide such a benchmark with community sizes and node degrees following power law distributions [LFR08; LF09a]. Variants of it also provide directed and/or weighted edges and possibly overlapping communities [LF09a]. Today, the LFR benchmark is widely used for the evaluation of community detection algorithms [FH16; Emm+16]. We describe the LFR benchmark in more detail in Section 2.1.1.

For overlapping communities, the typical parametrization of the LFR benchmark uses a quite simple model, though: a parameter determines the number of nodes that belong to more than one community, and another parameter determines the number of communities each of these nodes is part of. Studies on networks with communities defined by metadata show, however, that the number of communities per node rather follows a power law distributed instead of being uniform [YL12a; YL14]. They model each community as an Erdős-Rényi random graph and show that this AGM model [YL12a; YL14] that is fitted to a given graph can capture many properties of the given graph like clustering coefficients. The CKB generator [Chy+14] is based on the AGM model, but instead of fitting the model to a given graph, it describes a random graph model where community sizes as well as the number of communities per node follow power law distributions. Communities are generated as Erdős-Rényi random graphs with decreasing density for larger communities as also observed with the AGM model [YL14]. In Section 2.1.2, we describe the CKB benchmark in more detail.

Recently, the distributed A-BTER benchmark has been proposed [Slo+19] that we discuss in Section 2.1.3 which has some similarities with LFR, but is definitely not the same model. In Section 2.1.4, we discuss the nPSO model [MC18] that sounds interesting but has not received much attention until now.

2.1.1 LFR

This description of the LFR benchmark is based on our description of the LFR benchmark in our work on local community detection [HRW17] and our survey on network generation [Pen+20]. The first is joint work with Eike Röhrs and Dorothea Wagner while the second is joint work with Manuel Penschuck, Ulrik Brandes, Sebastian Lamm, Ulrich Meyer,Ilya Safro, Peter Sanders and Christian Schulz.

The LFR benchmark has been first introduced for unweighted, undirected graphs [LFR08] and later extended to directed and weighted graphs with (possibly) overlapping communities [LF09a]. The node degrees are drawn from a power law distribution with user-supplied parameters; community sizes are drawn from an independent power law distribution. The parameter μ_t determines the fraction of neighbors of every node u that are not part of u's communities. The remainder of a node's degree, its internal degree, is divided evenly among all communities it belongs to. For weighted graphs, the parameter μ_w determines the fraction of the weighted degree that is distributed among neighbors of u that are not part of u's community while the remainder is evenly divided among all communities u belongs to. Hence, μ_t is referred to as the topological mixing parameter and μ_w as the weight mixing parameter. If $\mu_t = \mu_w$, the edge weight is on average 1.0 for intra- and inter-community edges. In the case of unweighted graphs, we will frequently just use μ to refer to μ_t Nodes are assigned to communities at random such that each node's internal degree is smaller than the size of the community as otherwise not enough neighbors can be found. For overlapping communities, O_m specifies the number of communities every node is part of. For each community, LFR generates a random graph with the given internal degrees. It analogously samples the global *inter-community* graph with the remaining degrees. In an additional step, it ensures that no edges of the global graph are within a community and that no edge is part of multiple communities. In Chapter 3 we propose an algorithm to generate graphs following the LFR benchmark model larger than the RAM of a single machine using external memory in the form of SSDs.

2.1.2 CKB

The CKB benchmark graphs are inspired by the AGM model [YL12a]. In the AGM model, a community is a $G(n, p)$ graph, i.e., a graph of n nodes where every edge exists with probability p [Gil59]. The authors show [YL12a] that overlapping communities of this type may explain many properties observed in real-world networks when the number of communities the nodes are part of follows a power law distribution. In the CKB generator [Chy+14], the authors explicitly generate such a community structure. They draw community sizes as well as the number of communities per node from user-supplied power law distributions. The density of the communities decreases with increasing size of the communities. A global epsilon community adds additional noise to the graph. The authors implemented the CKB benchmark as a distributed algorithm using Apache Spark [Zah+16]. In Chapter 4, we propose a dynamic version of the CKB model.

2.1.3 A-BTER

This description of A-BTER has been taken from our survey on network generation [Pen+20] that is joint work with Manuel Penschuck, Ulrik Brandes, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders and Christian Schulz.

A-BTER [Slo+19] (Adapted BTER) uses a specially configured version of BTER [Kol+14] to generate benchmark graphs for community detection similar to the LFR benchmark. At the same time, A-BTER replicates the degree and clustering coefficient distribution of a given graph similarly to BTER. Using a heuristic scaling mechanism, A-BTER slightly adapts these distributions to match a prescribed average and maximum degree and clustering coefficient on a possibly larger graph. A-BTER takes a mixing parameter that, like in the LFR model, denotes the fraction of inter-cluster edges. Using a linear program, A-BTER further adjusts the degree and clustering coefficient distributions such that the resulting graph matches the mixing parameter while keeping the adjustments minimal. The assignment to affinity blocks, and the edge generation closely follows to BTER model. Using the parallel edge-skipping technique [Ala+16], the actual intra- and inter-community edges are generated efficiently in a parallel, distributed implementation. The implementation is based on MPI and OpenMP. On 512 compute nodes where each node has 128 GB RAM and two marvel ThunderX2 ARM processors with 28 cores per processor, A-BTER generates a graph with 925 billion edges and 4.6 billion nodes in 76 seconds. While the resulting graphs do not perfectly match the original LFR model, they show that community detection algorithms show similar behavior. Further, they show that both the degree and clustering coefficient distributions and the mixing parameter are as desired.

2.1.4 nPSO

The nonuniform popularity-similarity-optimization (nPSO) model [MC18] distributes points across the hyperbolic plane according to a mixture of Gaussians. Each node is connected to its k closest neighbors, neighbors are selected with decreasing probability. The authors report running times of a minute for 1000 nodes, indicating that the current implementation is not very scalable. The authors present some experiments showing that the Louvain algorithm (see Section 1.4.1) is capable of detecting the communities at least with some configurations. Further experiments would be necessary to determine its suitability as a benchmark. Different models of hyperbolic random graphs have been shown to model many features observed in real-world networks [Kri+10]. Thus, the nPSO model might be an interesting candidate for a benchmark that possibly not only features a community structure but also more realistic, non-trivial structures within and between communities that deviate from simpler random models like the LFR or the CKB benchmark.

2.2 Benchmark Instances

One of the most famous benchmark instances is the Karate graph of Zachary [Zac77], a small social network (34 nodes, 78 edges) of the people at a Karate club that split into two. Other famous benchmark instances include the “lesmis”, the dolphins and the football network. The nodes of the “lesmis” network are the characters of the novel “Les Misérables” by Victor Hugo, two nodes are connected if they appear in a chapter together [Knu93]. The dolphins network is a network of a bottlenose dolphin community living off Doubtful Sound, a fjord in New Zealand, where two nodes are connected if two dolphins spent significant time together [Lus+04]. The football network contains football clubs and their matches in one season [GN02]. All of them are rather small, the football network is the largest with 115 nodes and 613 edges. Thus, they are not suitable for the evaluation of scalable community detection algorithms. We use these small graphs only for the evaluation of our quasi-threshold editing algorithms in Chapter 9 and 10, as we compare to results published for them in [NG13]. Here, we also include the less commonly used “grass_web” network that models the food chain of grassland species [DHC95] as it has also been included in [NG13].

In the remaining parts, we use web graphs, a set of larger social networks from the SNAP collection and the Facebook 100 data set. We introduce them in the following sections.

2.2.1 Web Graphs

As part of the 10th DIMACS implementation challenge on graph partitioning and graph clustering [Bad+13], a larger benchmark set has been published [Bad+14]. Apart from the aforementioned small instances and many other networks, it also contains a set of larger web graphs [Bol+04; Bol+11; BV04]. Each node corresponds to a URL that has been discovered during a web crawl that has been restricted to a certain top-level domain that is indicated in the name of the graph. Edges represent links between the web pages at these URLs. For none of these graphs, any reference structure is known. While the URL of each node is available, we are not aware that this data is commonly used to evaluate community detection algorithms. The largest web graph that has been included in the 10th DIMACS challenge is the uk-2007-05 graph with about 100 million nodes and 3 billion edges. This makes them interesting for our work on scalable community detection. For our scalable editing heuristic in Chapter 9, we use four of these web graphs with up to 260 million edges while for our distributed community detection algorithm in Chapter 6, we also use the largest graph uk-2007-05.

2.2.2 SNAP

Another source of frequently used benchmark instances in community detection is the SNAP collection [LK14]. In particular, it contains a set of networks with reference community structure [YL12b]. We use the five undirected networks LiveJournal, Friendster, Orkut, Youtube, DBLP and Amazon. LiveJournal, Friendster, Orkut and Youtube are social networks where user can declare friendships which are represented as edges. Each of them also has groups users can join. Those are the reference communities. The DBLP network is a co-authorship network, i.e., each node represents an author and two nodes are connected if they published a paper together. Here, the reference communities are publication venues, i.e., each conference or journal forms a community. In the Amazon network, nodes are products and edges are between products that have been frequently purchased together. The reference communities are the product categories that are provided by Amazon. For all networks, reference communities consisting of multiple connected components have been split into their connected components and reference communities with less than 3 nodes have been removed. While it is claimed that those communities are "ground-truth communities", most of these communities cannot be recovered from the structure of the networks [HDF14; Har+14; FH16]. Therefore, we do not compare to these reference communities in our experiments.

2.2.3 Facebook 100

This introduction has been adapted from our paper on local community detection [HRW17] that is joint work with Eike Röhrs and Dorothea Wagner.

The Facebook 100 data set was first published by Traud et al. [TMP12]. It consists of 100 graphs, which are snapshots from September 2005. Each of these graphs describes the social network of Facebook at a university or college and is restricted to that university/college. The graph of a university or college consists of the persons who work, study, or studied there. Most of the nodes represent students. Unweighted edges between persons (nodes) exist, if they are friends on Facebook. Further, the nodes have attributes, like graduation year, dormitory, major and high school. While the smallest graph has just 769 nodes and 16 thousand edges, the largest graphs have up to 41 thousand nodes and almost 1.6 million edges. It has been shown that at least for some of the graphs there are correlations between edges and the attributes dormitory and graduation year [TMP12].

Visualizations [NOB15] also suggest that these attributes are represented in the structure of some of the networks. The Facebook networks are therefore frequently used in testing and comparing algorithms on real world data by looking at the correlations with their attributes [Lee+10; LC13; SMM14]. While it is clear that both certainly play a role in the organization of students, they are also clearly not the only factors. In our experiments on local community detection in Chapter 11, we also use this dormitory structure as a comparison while in our experiments on sparsification in Chapter 7, we only use the network structure. As also our experiments show, for a few networks we can approximately recover this structure while for others this is not the case.

2.3 Comparing Communities

To compare communities, we use three measures: the popular normalized mutual information, the adjusted rand index and the F_1 score. In the following we introduce these three measures. For an overview of other proposed measures we refer to surveys [WW07; Cha+17].

2.3.1 Normalized Mutual Information

Normalized mutual information (NMI) is the most frequently used comparison measure. The mutual information $\mathrm{MI}(\mathcal{C}, \mathcal{D})$ between two clusterings $\mathcal{C}$, $\mathcal{D}$ is:

$$\mathrm{MI}(\mathcal{C}, \mathcal{D}) := \sum_{C \in \mathcal{C}} \sum_{D \in \mathcal{D}} \frac{|C \cap D|}{n} \log \frac{n \cdot |C \cap D|}{|C||D|}$$

For NMI, this mutual information is normalized by the entropy H of $\mathcal{C}$ and $\mathcal{D}$. The entropy of a clustering $\mathcal{C}$ is

$$H(\mathcal{C}) = -\sum_{C \in \mathcal{C}} \frac{|C|}{n} \log \frac{|C|}{n}$$

The NMI can then be defined as:

$$\mathrm{NMI}(\mathcal{C}, \mathcal{D}) := \frac{\mathrm{MI}(\mathcal{C}, \mathcal{D})}{0.5 \cdot (H(\mathcal{C}) + H(\mathcal{D}))}$$

Note that there are several possibilities for the normalization, in particular instead of the average also the maximum of the two entropy values can be used [Kva87]. A Problem of NMI is that random clusterings with a high number of clusters can get non-zero scores, even above 0.5, when comparing to the ground truth [VEB09; AP15].

For overlapping communities, several generalizations of NMI have been proposed [LFK09; MGH11; VR12]. The first of them has been shown to have strange behavior, e.g., if one compared clustering consists of a single cluster that perfectly corresponds to a cluster in the other clustering, the score will be 0.5 [MGH11]. In this thesis, we use the one proposed by Esquivel and Rosvall [VR12], but only to compare clusterings found on instances generated by several implementations of the LFR benchmark, not to actually evaluate community detection algorithms.

2.3.2 Adjusted Rand Index

The similarity between two clusterings $\mathcal{C}$ and $\mathcal{D}$ can also be defined by comparing node pairs. Let a be the number of pairs of nodes that are in both clusterings together in a cluster. Similarly, let b be the number of pairs of nodes that are in both clusterings separated. The remaining node pairs are in one clustering separated and in the other together, let the number of these pairs be c. Then the rand measure R is defined as

$$R = \frac{a+b}{a+b+c} = \frac{a+b}{n(n-1)/2}$$

As the rand index is not zero for two random clusterings, an adjustment for chance has been proposed [HA85]. The adjusted rand index is the normalized difference of the rand index and its expected value on a clusterings that are drawn randomly with the same number of clusters and elements in each cluster as $\mathcal{C}$ and $\mathcal{D}$. It is defined as follows:

$$\text{ARI} := \frac{\sum_{C\in\mathcal{C}}\sum_{D\in\mathcal{D}}\binom{|C\cap D|}{2} - t_3}{0.5(t_1+t_2)-t_3}$$

where $t_1 = \sum_{C\in\mathcal{C}}\binom{|C|}{2}$, $t_2 = \sum_{D\in\mathcal{D}}\binom{|D|}{2}$ and $t_3 = t_1 t_2/\binom{n}{2}$ [WW07].

Note that while the ARI is zero in expectation for random clusterings, it can also be negative. The maximum value of 1 indicates that both clusterings are identical.

2.3.3 F_1 Score

When comparing just a single found community to a single ground truth community, we are basically evaluating a binary classifier. The F_1 score, the harmonic mean of precision and recall, is a frequently used measure of such a classifier's accuracy. Given a found community C and a ground truth community D, the precision is:

$$\text{precision}(C,D) := \frac{|C\cap D|}{|C|}$$

while the recall is:

$$\text{recall}(C,D) := \frac{|C\cap D|}{|D|}$$

The F_1 score is then

$$F_1(C,D) := 2\cdot\frac{\text{precision}(C,D)\cdot\text{recall}(C,D)}{\text{precision}(C,D)+\text{recall}(C,D)}$$

If we compare a found community to a (possibly overlapping) set of ground truth communities, we compare it to the best-matching ground truth community, i.e., given a found community C and a set of ground truth communities $\mathcal{D}$, we define:

$$F_1(C, \mathcal{D}) := \max_{D \in \mathcal{D}} F_1(C, D)$$

This can be extended to a set of found communities by using the weighted average:

$$F_w(\mathcal{C}, \mathcal{D}) = \frac{1}{\sum_{C \in \mathcal{C}} |C|} \sum_{C \in \mathcal{C}} |C| F_1(C, \mathcal{D})$$

This score only tells us if every found community has a well-matching ground truth community but not if every ground truth community also has a well-matching found community. To measure both, we use the *average weighted F1 score*, which is the average of $F_w(C_G, C_D)$ and $F_w(C_D, C_G)$. A similar score has been used in previous work [YL12a], but they used the unweighted average. We instead use the weighted average to give larger communities, which are typically underrepresented due to the power law distributions used in benchmark generators, a larger influence on the score. This is similar to [LA99], though they only considered one direction and not the average of both.

3 I/O-Efficient Generation of Massive Graphs Following the *LFR* Benchmark

This chapter is based on joint work with Ulrich Meyer, Manuel Penschuck, Hung Tran and Dorothea Wagner [Ham+18b]. Parts of it have been published as [Ham+17]. Compared to the publication, this chapter has been slightly adapted to reference parts of this thesis where appropriate and we added a discussion of follow-up work on Curveball trades [Car+18] to the outlook. We thank the anonymous reviewers for their insightful comments and recommendations and Hannes Seiwert and Mark Ortmann for valuable discussions on *EM-HH*.

We present *EM-LFR*, the first external memory algorithm able to generate massive complex networks following the *LFR* benchmark. Its most expensive component is the generation of random graphs with prescribed degree sequences which can be divided into two steps: the graphs are first materialized deterministically using the Havel-Hakimi algorithm, and then randomized. Our main contributions are *EM-HH* and *EM-ES*, two I/O-efficient external memory algorithms for these two steps. We also propose *EM-CM/ES*, an alternative sampling scheme using the Configuration Model and rewiring steps to obtain a random simple graph. In an experimental evaluation we demonstrate their performance; our implementation is able to handle graphs with more than 37 billion edges on a single machine, is competitive with a massively parallel distributed algorithm, and is faster than a state-of-the-art internal memory implementation even on instances fitting in main memory. *EM-LFR*'s implementation is capable of generating large graph instances orders of magnitude faster than the original implementation. We give evidence that both implementations yield graphs with matching properties by applying clustering algorithms to generated instances. Similarly, we analyze the evolution of graph properties as *EM-ES* is executed on networks obtained with *EM-CM/ES* and find that the alternative approach can accelerate the sampling process.

3.1 Introduction

With the emergence of massive networks that cannot be handled in the main memory of a single computer, new clustering schemes have been proposed for advanced models of computation [Buz+14; ZY16]. Since such algorithms typically use hierarchical input representations, quality results of small benchmarks may not be generalizable to larger instances [Emm+16; Ham+18a], see also Chapter 6. Often though, the quality is only evaluated on small benchmark graphs as currently available graph clustering benchmark generators are unable to generate the necessary graphs [BH15; Buz+14]. Instead, computationally inexpensive random graph models such as RMAT are used [Que+13] to generate huge graphs. Using those models, it is however not possible to evaluate whether the clustering algorithm is actually able to detect communities on such a large graph as there is no ground truth community structure to compare against. Filling this gap, we propose a generator in the external memory (EM) model following the *LFR* benchmark in order to produce clustering benchmark graph instances exceeding main memory. We implement the variants of the *LFR* benchmark for unweighted, undirected graphs with either overlapping or non-overlapping communities. Our proposed graph benchmark generator has already been used to evaluate the clustering quality of distributed clustering algorithms on graphs with up to 512 million nodes and 76.6 billion edges [Ham+18a], see also Chapter 6.

The distributed CKB benchmark [Chy+14] is a step into a similar direction, however, it considers only overlapping clusters and uses a different model of communities. In contrast, our approach is a direct realization of the established *LFR* benchmark and supports both disjoint and overlapping clusters.

3.1.1 Random Graphs from a prescribed Degree Sequence

The *LFR* benchmark uses the *fixed degree sequence model* (FDSM), also known as edge switching Markov chain algorithm (e.g. [Mil+03]), to obtain a random graph following a previously computed degree sequence. In preliminary studies, we identified this task as the main issue when transferring the *LFR* benchmark into an EM setting; both in terms of algorithmic complexity and runtime.

FDSM consists of two steps, namely (i) generating a deterministic graph from a prescribed degree sequence and (ii) randomizing this graph using random edge switches. For each edge switch, two edges are chosen uniformly at random and two of the endpoints are swapped if the resulting graph is still simple (cf. section 3.6). Each edge switch can be seen as a transition in a Markov chain. This Markov chain is irreducible [EH80], symmetric and aperiodic [GMZ03] and therefore converges to the uniform distribution. It also has been shown to converge in polynomial time if the maximum degree is not too large compared to the number of edges [GS18]. However, the current analytical bounds of the mixing time are impractically high even for small graphs.

Experimental results on the occurrence of certain motifs in networks [Mil+03] suggest that $100m$ steps should be more than enough where m is the number of edges. Further results for random connected graphs [GMZ03] suggest that the average and maximum path length and link load converge between $2m$ and $8m$ swaps. More recently, further theoretical arguments and experiments showed that $10m$ to $30m$ steps are enough [RPS15].

A faster way to realize a given degree sequence is the *Configuration Model* which allows multi-edges and self-loops. In the *Erased Configuration Model* these illegal edges are deleted. Doing so, however, alters the graph properties and does not properly realize the skewed degree distributions required for *LFR* [SHZ15]. In this context the question arises whether edge switches starting from the Configuration Model can be used to uniformly sample simple graphs at random.

3.1.2 Our Contribution

We introduce *EM-LFR*[1], the first I/O-efficient *LFR* variant, and study the FDSM in the external memory model. After defining our notation, we summarize the original *LFR* benchmark in section 3.3. As illustrated in Figure 3.3, *EM-LFR* consists of several algorithmic building blocks which we discuss in sections 3.4 to 3.8. Here, the focus lies on FDSM consisting of (i) generating a deterministic graph from a prescribed degree sequence (cf. *EM-HH*, section 3.5) and (ii) randomizing this graph using random edge switches (cf. *EM-ES*, section 3.6). For *EM-HH*, we describe a streaming algorithm whose internal data structure only has an I/O complexity linear in the number of different degrees if a monotonous degree sequence is provided. To execute a number of edge switches proportional to the number m of edges, *EM-ES* triggers $\mathcal{O}(\text{sort}(m))$ I/Os. For *EM-LFR*, the I/O complexity is the same as it is dominated by the edge randomization step. In section 3.7, we additionally describe *EM-CM/ES*, an alternative to FDSM. It generates uniform random non-simple graphs using the Configuration Model in $\mathcal{O}(\text{sort}(m))$ I/Os and then obtains a simple graph by applying edge rewiring steps.

We conclude with an experimental evaluation of our algorithms and demonstrate that our EM version of the FDSM is faster than an existing state-of-the-art implementation even for instances still fitting into RAM. It scales well to large networks, as we demonstrate by handling a graph with 37 billion edges on a desktop computer, and almost an order of magnitude more efficient than an existing distributed parallel algorithm. Further, we compare *EM-LFR* to the original *LFR* implementation and show that *EM-LFR* is significantly faster while producing equivalent networks in terms of community detection algorithm performance and graph properties. A LFR benchmark graph with more than 1×10^{10} edges can be generated in 17 h on a single server with 64 GB RAM and 3 SSDs. We also investigate the mixing time of *EM-ES* and *EM-CM/ES* and give evidence that our alternative sampling scheme quickly yields uniform samples and that the number of swaps suggested by the original *LFR* implementation can be kept for *EM-LFR*.

3.2 Preliminaries and Notation

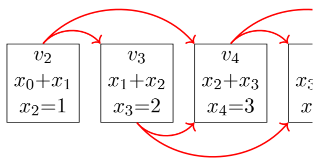
In this section, we briefly introduce the notation we use, and give an introduction to the external memory model as well as Time-Forward Processing, a crucial design-principle used in *EM-ES*.

3.2.1 Notation

We use the notation defined in Section 1.2. Unless stated differently, our EM algorithms represent a graph $G = (V, E)$ as a sequence containing for every ordered edge $[u, v] \in E$ only the entry (u, v).

Also refer to Table 3.1 which contains a summary of commonly used definitions.

[1]The implementation is freely available at `https://massive-graphs.org/extmem-lfr`. Amongst others, it contains encapsulated implementations of *EM-ES* and *EM-CM/ES* which can be easily reused for novel application.



```
// Send base cases of $x_0 = 0$ and $x_1 = 1$ to $v_2$
PQ.push(<key=2, value=0>, <key=2, value=1>)
for i ← 2, ..., n: // calculation of $v_i$ in $G$
    sum ← 0
    // will always fetch exactly two elements
    while (PQ.min.key == i): sum += PQ.remove_min().value
    // report value and send message to successors
    print($x_i$ = sum); PQ.push(<key=i+1, sum>, <key=i+2, sum>)
```

Figure 3.1: **Left:** Dependency graph of the Fibonacci sequence (ignoring base case). **Right:** Time Forward Processing to compute sequence.

3.2.2 External-Memory Model

In contrast to classic models of computation, such as the unit-cost RAM, modern computers contain deep memory hierarchies ranging from fast registers, caches and main memory to solid state drives (SSDs) and hard disks. Algorithms unaware of these properties may face performance penalties of several orders of magnitude. We use the commonly accepted external memory (EM) model by Aggarwal and Vitter [AV88] to reason about the influence of data locality in memory hierarchies. The model features two memory types with fast internal memory (IM) which may hold up to M data items, and a slow disk of unbounded size. The input and output of an algorithm are stored in EM while computation is only possible on values in IM. The measure of an algorithm's performance is the number of I/Os required. Each I/O transfers a block of B consecutive items between memory levels. Reading or writing n contiguous items from or to disk requires $\text{scan}(n) = \Theta(n/B)$ I/Os. Sorting n contiguous items uses $\text{sort}(n) = \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os. For realistic values of n, B and M, $\text{scan}(n) < \text{sort}(n) \ll n$. Sorting complexity often constitutes a lower bound for intuitively non-trivial tasks [AV88; MSS03].

3.2.3 *TFP*: Time Forward Processing

Time Forward Processing (*TFP*) is a generic technique to manage data dependencies of external memory algorithms [MZ03]. Consider an algorithm computing values $x_1, \ldots, x_n$ in which the calculation of x_i requires previously computed values. One typically models these dependencies using a directed acyclic graph $G{=}(V, E)$. Every node $v_i \in V$ corresponds to the computation of x_i, and an edge $(v_i, v_j) \in E$ indicates that the value x_i is necessary to compute x_j. As an example consider the Fibonacci sequence $x_0 = 0,\ x_1 = 1,\ x_i = x_{i-1}{+}x_{i-2}\ \forall i \geq 2$ in which each node v_i with $i \geq 2$ depends on exactly its two predecessors (cf to Figure 3.1). In this case, a linear scan for increasing i solves the dependencies.

In general, an algorithm needs to traverse G according to some topological order $\prec_T$ of nodes V and also has to ensure that each v_j can access values from all v_i with $(v_i, v_j) \in E$. The *TFP* technique achieves this as follows: as soon as x_i has been calculated, messages of form $\langle v_j, x_i \rangle$ are sent to all successors $(v_i, v_j) \in E$. These messages are kept in a minimum priority queue sorting the items by their recipients according to $\prec_T$. By definition, the algorithm only starts the computation v_i once all predecessors $v_j \prec_T v_i$ are completed. Since these predecessors already removed their messages from the PQ, items addressed to

Symbol	*Description*
$[k]$	$[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_+$ (section 1.2)
$[u, v]$	Undirected edge with implication $u \leq v$ (section 1.2)
$\langle X \rangle$	The mean $\langle X \rangle := \sum_{i=1}^{n} x_i/n$
$\langle X^2 \rangle$	The *second moment* $\langle X^2 \rangle := \sum_{i=1}^{n} x_i^2/n$
B	Number of items in a block transferred between IM and EM (section 3.2.2)
$d_{\min}$, $d_{\max}$	Min/max degree of nodes in *LFR* benchmark (section 3.3)
d_v^{in}	$d_v^{\text{in}} = (1-\mu)\cdot d_v$, intra-community degree of node v (section 3.3)
$\mathcal{D}$	$\mathcal{D} = (d_1, \dots, d_n)$ with $d_i \leq d_{i+1} \forall i$. Degree sequence of a graph (section 3.5)
$D(\mathcal{D})$	$D(\mathcal{D}) = \left\|\{d_i : 1 \leq i \leq n\}\right\|$ where $\mathcal{D} = (d_1, \dots, d_n)$, degree support (section 3.5)
n	Number of vertices in a graph (section 1.2)
m	Number of edges in a graph (section 1.2)
μ	Mixing parameter in *LFR* benchmark, i.e. ratio of neighbors that shall be in other communities (section 3.3)
M	Number of items fitting into internal memory (section 3.2.2)
$\textsc{Pld}\,([a, b), \gamma)$	Powerlaw distribution with exponent $-\gamma$ on the interval $[a, b)$ (section 1.2)
$s_{\min}$, $s_{\max}$	Min/max size of communities in *LFR* benchmark (section 3.3)
$\text{scan}(n)$	$\text{scan}(n) = \Theta(n/B)$ I/Os, scan complexity (section 3.2.2)
$\text{sort}(n)$	$\text{sort}(n) = \Theta((n/B) \cdot \log_{M/B}(n/B))$ I/Os, sort complexity (section 3.2.2)

Table 3.1: Definitions used in this chapter.

v_i (if any) are currently the smallest elements in the data structure and can be dequeued. Using a suited EM PQ [Arg95; San00], TFP incurs $\mathcal{O}(\text{sort}(k))$ I/Os, where k is the number of messages sent.

3.3 The *LFR* Benchmark

In this section we introduce the general approach for generating *LFR* benchmark graphs, outline important algorithmic challenges, and address each of them by proposing a suited EM algorithm in the following chapters (refer to Figure 3.3 for an overview).

As introduced in Section 2.1.1, the *LFR* benchmark [LFR08] describes a generator for random graphs featuring node degrees and community sizes both following powerlaw distributions. The produced networks also contain a planted community structure against which the performance of detection algorithms is measured. A revised version [LF09a] additionally introduces weighted and directed graphs with overlapping communities and changes the sampling algorithm even for the original settings. We consider the

Parameter	*Meaning*
n	Number of nodes to be produced
$\textsc{Pld}\,([d_{\min}, d_{\max}), \gamma)$	Degree distribution of nodes, typically $\gamma = 2$
$0 \le O \le n$, $\nu \ge 1$	O random nodes belong to ν communities; remainder has one membership
$\textsc{Pld}\,([s_{\min}, s_{\max}), \beta)$	Size distribution of communities, typically $\beta = 1$
$0 < \mu < 1$	Mixing parameter: fraction of neighbors of every node u that shall not share a community with u

Table 3.2: Parameters of overlapping *LFR*. The typical values follow suggestions by [LF09a].

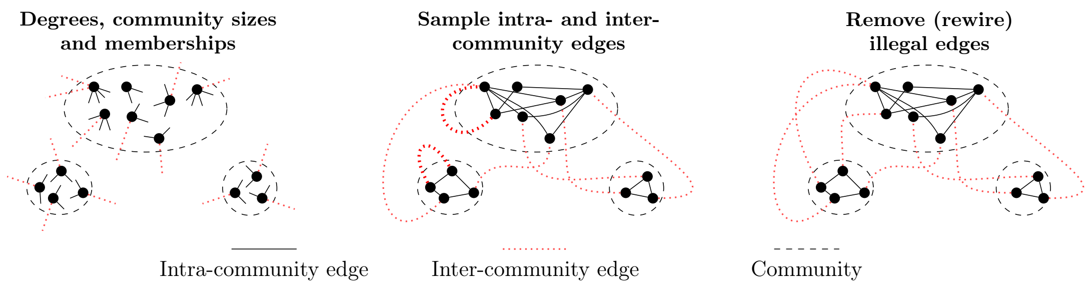

Degrees, community sizes and memberships
Sample intra- and inter-community edges
Remove (rewire) illegal edges
Intra-community edge
Inter-community edge
Community

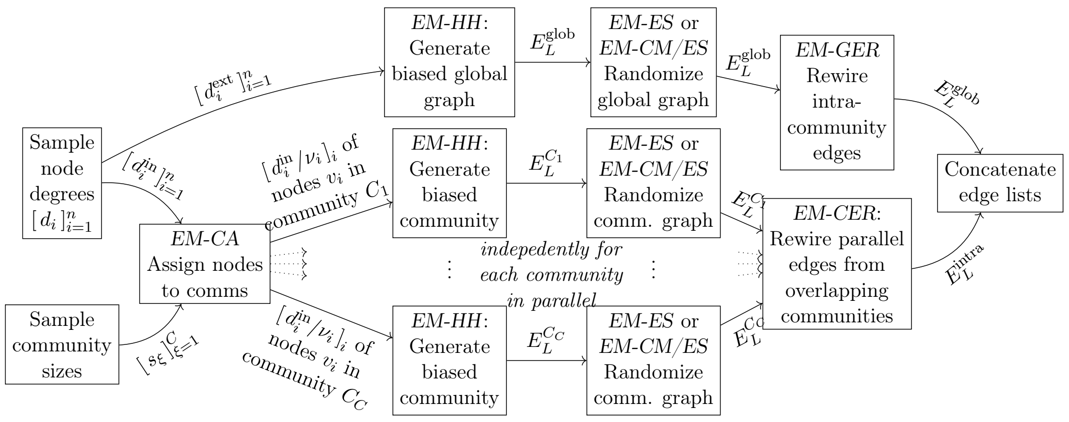
Figure 3.2: **Left:** Sample node degrees and community sizes from two powerlaw distributions. The mixing parameter μ determines the fraction of the inter-community edges. Then, assign each node to sufficiently large communities. **Center:** Sample intra-community graphs and inter-community edges independently. This may lead to illegal intra-community edges in the global graph as shown here in bold. **Right:** Lastly, remove illegal inter-community edges respective to the global graph.

modern generator, which is also used in the author's implementation, and focus on the most common variants for unweighted, undirected graphs and optionally overlapping communities. All its parameters are listed in Table 3.2 and are fully supported by *EM-LFR*.

LFR starts by randomly sampling the degrees $\mathcal{D} = [\, d_i \,]_{i=1}^{n}$ of all nodes, the numbers $[\, \nu_i \,]_{i=1}^{n}$ of clusters they are members in, and community sizes $S = [\, s_\xi \,]_{\xi=1}^{C}$ such that $\sum_{\xi=1}^{C} s_\xi = \sum_{i=1}^{n} \nu_i$ according to the supplied parameters. During this process the number of communities C follows endogenously and is bounded by $C{=}\mathcal{O}(n)$ even if nodes are members in $\nu{=}\mathcal{O}(1)$ communities.[2]

Depending on the *mixing parameter* $0 < \mu < 1$, every node $v_i \in V$ is incident to $d_i^{\text{ext}} = \mu \cdot d_i$ inter-community edges and $d_i^{\text{in}} = (1{-}\mu) \cdot d_i$ edges within its communities. In the case of overlapping communities, the internal degree is evenly split among all

[2]Under the realistic assumption that the maximal community size grows with $s_{\max} = \Omega(n^{\epsilon})$ for some $\epsilon > 0$, the bound improves to $C{=}o(n)$ with high probability due to the powerlaw distributed community sizes.



Sample node degrees $[d_i]_{i=1}^n$
Sample community sizes
$[s_\xi]_{\xi=1}^C$
$[d_i^{\text{ext}}]_{i=1}^n$
$[d_i^{\text{in}}]_{i=1}^n$
EM-CA Assign nodes to comms
$[d_i^{\text{in}}/\nu_i]_i$ of nodes v_i in community C_1
$[d_i^{\text{in}}/\nu_i]_i$ of nodes v_i in community C_C
EM-HH: Generate biased global graph
EM-HH: Generate biased community
E_L^{glob}
$E_L^{C_1}$
$E_L^{C_C}$
EM-ES or EM-CM/ES Randomize global graph
EM-ES or EM-CM/ES Randomize comm. graph
indepedently for each community in parallel
EM-GER Rewire intra-community edges
EM-CER: Rewire parallel edges from overlapping communities
E_L^{intra}
Concatenate edge lists

Figure 3.3: The *EM-LFR* pipeline: After randomly sampling the node degrees and community sizes, nodes are assigned into suited communities by *EM-CA* (section 3.4). The global (inter-community) graph and each community graph is then generated independently by first materializing biased graphs using *EM-HH* (section 3.5) followed by a randomization using *EM-ES* or *EM-CM/ES* (sections 3.6 and 3.7). The global graph may contain edges between nodes of the same community which would decrease the mixing μ and are hence rewired using *EM-GER* (section 3.8.1). Similarly, two overlapping communities can have identical edges which are rewired by *EM-CER* (section 3.8.2).

communities the node is part of. Both the computation of d_i^{in} and the division d_i^{in}/ν_i into several communities use non-deterministic rounding to avoid biases. *LFR* assigns every node v_i to either exactly $\nu_i = 1$ or $\nu_i = \nu$ communities at random such that the requested community sizes and number of communities per node are realized. It further ensures, that the desired internal degree d_i^{in}/ν_i is strictly smaller than the size s_ξ of its community ξ.

As illustrated in Figure 3.2, the *LFR* benchmark then generates the inter-community graph using FDSM on the degree sequence $[\, d_i^{\text{ext}} \,]_{i=1}^n$. In order not to violate the mixing parameter μ, rewiring steps are applied to the global inter-community graph to replace edges between two nodes sharing a community. Analogously, an intra-community graph is sampled for each community. In the overlapping case, rewiring steps may be necessary to remove edges that exist in multiple communities and would result in duplicate edges in the final graph.

3.4 *EM-CA*: Community Assignment

In the *LFR* benchmark, every node belongs to one (non-overlapping) or more (overlapping) communities. *EM-CA* finds such a random assignment subject to the two constraints that all communities get as many nodes as previously determined (see Figure 3.3) and that for a node v_i all its assigned communities have enough other members to satisfy the node's intra-community degree d_i^{in}/ν_i.

For the sake of simplicity, we first restrict ourselves to the non-overlapping case, in which every node belongs to exactly one community. Consider a sequence of community sizes $S = [\, s_\xi \,]_{\xi=1}^C$ with $n = \sum_{\xi=1}^C s_\xi$ and a sequence of intra-community degrees $\mathcal{D} = [\, d_i^{\text{in}} \,]_{i=1}^n$. Let S and $\mathcal{D}$ be non-decreasing and positive. The task is to find a random surjective assignment $\chi\colon V \to [C]$ with:

(R1) Every community ξ is assigned s_ξ nodes as requested, with $s_\xi := \big|\{v \mid v \in V \wedge \chi(v){=}\xi\}\big|$.

(R2) Every node $v \in V$ becomes member of a sufficiently large community $\chi(v)$ with $s_{\chi(v)} > d_v^{\text{in}}$.

Observe that χ can be interpreted as a bipartite graph where the partition classes are given by the communities $[C]$ and nodes $[\, v_i \,]_{i=1}^n$ respectively, and each edge corresponds to an assignment.

3.4.1 A simple, iterative, but not yet complete algorithm

To ease the description of the algorithm, let us also ignore (R2) for now and discuss the changes needed in section 3.4.2. Then the assignment graph can be sampled in the spirit of the Configuration Model (cf. section 3.7). To do so, we draw a permutation π of nodes uniformly at random and assign nodes $[\, v_{\pi(i)} \,]_{i=x_\xi+1}^{x_\xi+s_\xi}$ to community ξ where $x_\xi := \sum_{i=1}^{\xi-1} s_i$ is the number of slots required for communities with indices smaller than ξ.

To ease later modifications, we prefer an equivalent iterative formulation: while there exists a yet unassigned node u, draw a community X with probability proportional to the number of its remaining free slots (i.e. $\mathbb{P}[X{=}\xi] \propto s_\xi$). Assign node u to X, reduce the community's probability mass by decreasing $s_X \leftarrow s_X - 1$ and repeat. By construction, the first scheme is unbiased and the equivalence of both approaches follows as a special case of Lemma 3.1 (see below).

We implement the random selection process efficiently based on a binary tree where each community corresponds to a leaf with a weight equal to the number of free slots in the community. Inner nodes store the total weight of their left subtree. In order to draw a community, we sample an integer $Y \in [0, W_C)$ uniformly at random where $W_C := \sum_{\xi=1}^{C} s_\xi$ is the tree's total weight. Following the tree according to Y yields the leaf corresponding to community X. An I/O-efficient data structure [MP16] based on lazy evaluation for such dynamic probability distributions enables a fully external algorithm with $\mathcal{O}(n/B \cdot \log_{M/B}(C/B)) = \mathcal{O}(\text{sort}(n))$ I/Os. However, if $C < M$, we can store the tree in IM, allowing a semi-external algorithm which only needs to scan through $\mathcal{D}$, triggering $\mathcal{O}(\text{scan}(n))$ I/Os.

3.4.2 Enforcing constraint on community size (R2)

To enforce (R2), we additionally ensure that all nodes are assigned to a sufficiently large community such that they find enough neighbors to connect to. We exploit that S and $\mathcal{D}$ are non-decreasing and define $p_v := \max\{\xi \mid s_\xi > d_v^{\text{in}}\}$ as the index of the smallest community node v may be assigned to. Since $[\, p_v \,]_v$ is therefore monotonic itself, it can be computed online with $\mathcal{O}(1)$ additional IM and $\mathcal{O}(\text{scan}(n))$ I/Os in the fully external setting by scanning through S and $\mathcal{D}$ in parallel. In order to restrict the random sampling to the communities $\{1, \ldots, p_v\}$, we reduce the aforementioned random interval to $[0, W_v)$ where the partial sum $W_v := \sum_{\xi=1}^{p_v - 1} s_\xi$ is available while computing p_v. We generalize the notation of uniformity to assignments subject to (R2) as follows:

Lemma 3.1. *Given $S = [\, s_\xi \,]_{\xi=1}^{C}$ and $\mathcal{D}$, let $u, v \in V$ be two nodes with the same constraints $p_u = p_v$ and let c be an arbitrary community. Further, let χ be an assignment generated by EM-CA. Then, $\mathbb{P}[\chi(u){=}c] = \mathbb{P}[\chi(v){=}c]$.*

Proof. Without loss of generality, assume that $p_u = p_1$, i.e. u is one of the nodes with the tightest constraints. If this is not the case, we just execute *EM-CA* until we reach a node u' which has the same constraints as u does (i.e. $p_{u'} = p_u$), and apply the Lemma inductively. This is legal since *EM-CA* streams through $\mathcal{D}$ in a single pass and is oblivious to any future values. In case $c > p_1$, neither u nor v can become a member of c. Therefore, $\mathbb{P}[\chi(u){=}c] = \mathbb{P}[\chi(v){=}c] = 0$ and the claim follows trivially.

Now consider the case $c \le p_1$. Let $s_c^{(i)}$ be the number of free slots in community c at the beginning of round $i \ge 1$ and $W^{(i)} = \sum_{j=1}^{C} s_j^{(i)}$ their sum at that time. By definition, *EM-CA* assigns node u to community c with probability $\mathbb{P}[\chi(u){=}c] = s_c^{(u)}/W^{(u)}$. Further, the algorithm has to update the number of free slots. Thus, initially we have $s_c^{(1)} = s_c$

and for iteration $1 < i \leq n$ it holds that

$$s_c^{(i)} = \begin{cases} s_c^{(i-1)} - 1 & \text{if } v_{i-1} \text{ was assigned to } c \\ s_c^{(i-1)} & \text{otherwise} \end{cases}.$$

The number of free slots is reduced by one in each step $W^{(i)} = W^{(1)} - i + 1 = \left(\sum_{j=1}^{C} S_j\right) - i + 1$.
The claim follows by transitivity if we show $\mathbb{P}[\chi(u){=}c] = s_c^{(u)}/W^{(u)} = s_c^{(1)}/W^{(1)}$. For $u = 1$ it holds by definition. Now, consider the induction step for $u{>}1$:

$$\begin{aligned}
\mathbb{P}[\chi(u){=}c] = \frac{s_c^{(u)}}{W^{(u)}} &= \mathbb{P}[\chi(u{-}1){=}c]\frac{s_c^{(u-1)} - 1}{W^{(u)}} + \mathbb{P}[\chi(u{-}1){\neq}c]\frac{s_c^{(u-1)}}{W^{(u)}} \\
&= \frac{s_c^{(u-1)}}{W^{(u-1)}}\frac{s_c^{(u-1)} - 1}{W^{(u)}} + \left(1 - \frac{s_c^{(u-1)}}{W^{(u-1)}}\right)\frac{s_c^{(u-1)}}{W^{(u)}} = \frac{s_c^{(u-1)} \cdot W^{(u-1)} - s_c^{(u-1)}}{W^{(u-1)} \cdot W^{(u)}} \\
&= \frac{s_c^{(u-1)}(W^{(u-1)} - 1)}{W^{(u-1)} \cdot (W^{(u-1)} - 1)} = \frac{s_c^{(u-1)}}{W^{(u-1)}} \overset{\text{Ind. Hyp.}}{=} \frac{s_c^{(1)}}{W^{(1)}} \qquad \square
\end{aligned}$$

3.4.3 Assignment with overlapping communities

In the overlapping case, the weight of S increases to account for nodes with multiple memberships. There is further an additional input sequence $[\,\nu_i\,]_{i=1}^{n}$ corresponding to the number of memberships node v_i shall have, each of which has d_i^{in}/ν_i intra-community neighbors. We then sample not only one community per node v_i, but ν_i different ones.

Since the number of memberships $\nu_v \ll M$ is small, a duplication check during the repeated sampling is easy in the semi-external case and does not change the I/O complexity. However, it is possible that near the end of the execution there are less free communities than memberships requested. We address this issue by switching to an offline strategy for the last $\Theta(M)$ assignments and keep them in IM. As $\nu = \mathcal{O}(1)$, there are $\Omega(\nu)$ communities with free slots for the last $\Theta(M)$ vertices and a legal assignment exists with high probability. The offline strategy proceeds as before until it is unable to find ν different communities for a node. In that case, it randomly picks earlier assignments until swapping the communities is possible.

In the fully external setting, the I/O complexity grows linearly in the number of samples taken and is thus bounded by $\mathcal{O}(\nu\,\text{sort}(n))$. However, the community memberships are obtained lazily and out-of-order which may assign a node several times to the same community. This corresponds to a multi-edge in the bipartite assignment graph. It can be removed using the rewiring technique detailed in section 3.7.2.

3.5 *EM-HH*: Deterministic Edges from a Degree Sequence

In this section, we address the issue of generating a graph from prescribed degrees and introduce an EM-variant of the well-known Havel-Hakimi scheme. It takes a positive non-decreasing degree sequence $\mathcal{D} = [\,d_i\,]_{i=1}^{n}$ and, if possible, outputs a graph $G_{\mathcal{D}}$ realizing

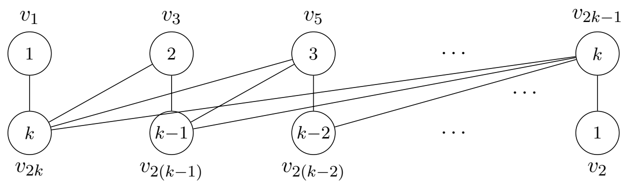


Figure 3.4: Materialization of the degree sequence $\mathcal{D}_k = (1,1,2,2,\ldots,k,k)$ with $D(\mathcal{D}_k) = k = \Theta(n)$ which maximizes *EM-HH*'s memory consumption asymptotically. A node's label corresponds to its degree.

these degrees.[3] *EM-LFR* uses this algorithm (cf. Figure 3.3) to first obtain a legal but biased graph following $\mathcal{D}$ and then randomizes $G_\mathcal{D}$ in a subsequent step.

A sequence $\mathcal{D}$ is called *graphical* if a matching simple graph $G_\mathcal{D}$ exists. Havel and Hakimi independently gave inductive characterizations of graphical sequences which directly lead to a graph generator [Hav55; Hak62]: given $\mathcal{D}$, connect the first node v_1 with degree d_1 (minimal among all nodes) to d_1-many high-degree vertices by emitting edges $\{\, \{v_1, v_{n-i}\} \mid 0 \le i < d_1 \,\}$. Then obtain an updated sequence $\mathcal{D}'$ by removing d_1 from $\mathcal{D}$ and decrementing the remaining degree of every new neighbor $\{\, v_{n-i} \mid 0 \le i < d_1 \,\}$.[4] Subsequently, remove zero-entries and sort $\mathcal{D}'$ while keeping track of the original positions to be able to output the correct node indices. Finally, recurse until no more positive entries remain. After every iteration, the size of $\mathcal{D}$ is reduced by at least one resulting in $\mathcal{O}(n)$ rounds.

For an implementation, it is non-trivial to keep the sequence ordered after decrementing the neighbors' degrees. Internal memory solutions typically employ priority queues optimized for integer keys, such as bucket-lists [SSM16; VL16]. This approach incurs $\Theta(\text{sort}(n+m))$ I/Os using a naïve EM PQ since every edge triggers an update to the pending degree of at least one endpoint.

We hence propose the Havel-Hakimi variant *EM-HH* which, for virtually all realistic powerlaw degree distributions, avoids accesses to disk besides writing the result. The algorithm emits a stream of edges in lexicographical order which can be fed to any single-pass streaming algorithm without a round-trip to disk. Thus, we consider only internal I/Os and emphasize that storing the output – if necessary by the application – requires $\mathcal{O}(m)$ time and $\mathcal{O}(\text{scan}(m))$ I/Os where m is the number of edges produced. Additionally, *EM-HH* may be used to test in time $\mathcal{O}(n)$ whether a degree sequence $\mathcal{D}$ is graphical or to drop problematic edges yielding a graphical sequence (cf. section 3.7).

3.5.1 Data structure

Instead of maintaining the degree of every node in $\mathcal{D}$ individually, *EM-HH* compacts nodes with equal degrees into a group, yielding $D(\mathcal{D}) := \big|\{d_i : 1{\le}i{\le}n\}\big|$ groups. Since $\mathcal{D}$ is monotonic, such nodes have consecutive ids and the compaction can be performed in a streaming fashion.[5] The sequence is then stored as a doubly linked list $L = [g_j]_{1\leq j\leq D(\mathcal{D})}$ where group $g_j = (b_j, n_j, \delta_j)$ encodes that the n_j nodes $[\,v_{b_j+i}\,]_{i=0}^{n_j-1}$ have degree δ_j. At the beginning of every iteration of *EM-HH*, L satisfies the following invariants which guarantee a compact representation:

(I1) The groups contain strictly increasing degrees, i.e. $\delta_j < \delta_{j+1} \;\; \forall 1 \leq j < |L|$

(I2) There are no gaps in the node ids, i.e. $b_j + n_j = b_{j+1} \;\; \forall 1 \leq j < |L|$

These invariants allow us to bound the memory footprint in two steps: first observe that a list L of size $D(\mathcal{D})$ describes a graph with at least $\sum_{i=1}^{D(\mathcal{D})} i/2$ edges due to (I1). Thus, materializing an arbitrary L of the main memory size $|L| = \Theta(M)$ emits $\Omega(M^2)$ edges. With as little as 2 GB RAM, this amounts to an edge list exceeding 1 PB in size.[6] Therefore, even in the worst case the whole data structure can be kept in IM for all practical scenarios. On top of this, a probabilistic argument applies: While there exist graphs with $D(\mathcal{D}) = \Theta(n)$ as illustrated in Fig 3.4, Lemma 3.2 gives a sub-linear bound on $D(\mathcal{D})$ if $\mathcal{D}$ is sampled from a powerlaw distribution.

Lemma 3.2. *Let $\mathcal{D}$ be a degree sequence with n nodes sampled from* $\textsc{Pld}\,([1,n),\gamma)$. *Then, the number of unique degrees $D(\mathcal{D}) = \big|\{d_i : 1 \leq i \leq n\}\big|$ is bounded by $\mathcal{O}(n^{1/\gamma})$ with high probability.*

Proof. Consider random variables $(X_1, \ldots, X_n)$ sampled i.i.d. from $\textsc{Pld}\,([1,n),\gamma)$ as an unordered degree sequence. Fix an index $1{\le}j{\le}n$. Due to the powerlaw distribution, X_j is likely to have a small degree. Even if all degrees $1, \ldots, n^{1/\gamma}$ were realized, their occurrences would be covered by the claim. Thus, it suffices to bound the number of realized degrees larger than $n^{1/\gamma}$.

We first show that their total probability mass is small. Then we can argue that $D(\mathcal{D})$

[3] *EM-LFR* directly generates a monotonic degree sequence by first sampling a monotonic uniform sequence [BS80; Vit87] and then applying the inverse sampling technique (carrying over the monotonicity) for a powerlaw distribution. Thus, no additional sorting steps are necessary for the inter-community graph.

[4] This variant is due to [Hak62]; in [Hav55], the node of maximal degree is picked and connected.

[5] While direct sampling of the group's multinomial distribution is not beneficial in *LFR*, it may be used to omit the compaction phase for other applications.

[6] A single item of L can be naïvely represented by its three values and two pointers, i.e. a total of $5{\cdot}8 = 40$ bytes per item (assuming 64 bit integers and pointers). Just 2 GB of IM suffice for storing 5×10^7 items, which result in at least 6.25×10^{14} edges, i.e. storing just two bytes per edge would require more than one Petabyte. Observe that standard tricks, such as exploiting the redundancy due to (I2), allow to reduce the memory footprint of L.

is asymptotically unaffected by their rare occurrences:

$$\begin{aligned}
\mathbb{P}[X_j{>}n^{1/\gamma}] &= \sum_{i=n^{1/\gamma}+1}^{n-1} \mathbb{P}[X_j{=}i] = \frac{\sum_{i=n^{1/\gamma}+1}^{n-1} i^{-\gamma}}{\sum_{i=1}^{n-1} i^{-\gamma}} \overset{(i)}{=} \frac{\sum_{i=n^{1/\gamma}+1}^{n-1} i^{-\gamma}}{\zeta(\gamma) - \sum_{i=n}^{\infty} i^{-\gamma}} \\
&\overset{(ii)}{\leq} \frac{\int_{n^{1/\gamma}}^{n-1} x^{-\gamma}\,\mathrm{d}x}{\zeta(\gamma) - \int_n^\infty x^{-\gamma}\,\mathrm{d}x} = \frac{\frac{1}{1-\gamma}\left[(n{-}1)^{1-\gamma} - n^{1/\gamma}/n\right]}{\zeta(\gamma) + \frac{1}{1-\gamma} n^{1-\gamma}} \\
&= \frac{n^{1/\gamma}/n - (n-1)^{1-\gamma}}{(\gamma-1)\zeta(\gamma) - n^{1-\gamma}} = \mathcal{O}(n^{1/\gamma}/n),
\end{aligned}$$

where (i) $\zeta(\gamma) = \sum_{i=1}^{\infty} i^{-\gamma}$ is the Riemann zeta function which satisfies $\zeta(\gamma) \geq 1$ for all $\gamma \in \mathbb{R}$, $\gamma \geq 1$. In step (ii), we exploit the series' monotonicity to bound it in between the two integrals $\int_a^{b+1} x^{-\gamma}\,\mathrm{d}x \leq \sum_{i=a}^{b} i^{-\gamma} \leq \int_{a-1}^{b} x^{-\gamma}\,\mathrm{d}x$.

In order to bound the number of occurrences, define Boolean indicator variables Y_i with $Y_i = 1$ iff $X_i{>}n^{1/\gamma}$ and observe that they model Bernoulli trials $Y_i \in B(p)$ with $p = \mathcal{O}(n^{1/\gamma}/n)$. Thus, the expected number of high degrees is $\mathbb{E}[\sum_{i=1}^{n} Y_i] = \sum_{i=1}^{n} \mathbb{P}[X_i{>}n^{1/\gamma}] = \mathcal{O}(n^{1/\gamma})$. Chernoff's inequality gives an exponentially decreasing bound on the tail distribution of the sum which thus holds with high probability. □

Remark. Experiments in section 3.10.2 indicate that the hidden constants in Lemma 3.2 are small for realistic γ.

Corollary 3.3. *With high probability graphs with $m = \mathcal{O}(M^{2\gamma})$ following a powerlaw degree distribution are processed without I/O.*

Proof. Due to Lemma 3.2 the number of unique degrees $D(\mathcal{D})$ is bounded by $\mathcal{O}(n^{1/\gamma})$ with high probability. Consequently, a list of size $D(\mathcal{D})$ filling the whole IM supports $n = \mathcal{O}(M^\gamma)$ many nodes and thus $m = \mathcal{O}(M^{2\gamma})$ many edges with high probability. □

3.5.2 Algorithm

EM-HH works in n rounds, where every iteration corresponds to a recursion step of the original formulation. Each time it extracts node v_{b_1} with the smallest available id and with minimal degree δ_1. The extraction is achieved by incrementing the lowest node id ($b_1' \leftarrow b_1{+}1$) of group g_1 and decreasing its size ($n_1' \leftarrow n_1{-}1$). If the group becomes empty ($n_1' = 0$), it is removed from L at the end of the iteration; Figure 3.5 illustrates this situation in step 2. We now connect node v_{b_1} to δ_1 nodes from the end of L. Let g_j be the group of smallest index to which v_{b_1} connects to.
Then there are two cases:

(C1) If node v_{b_1} connects to all nodes in g_j, we directly emit the edges $\big\{[v_{b_1}, x] \,|\, n{-}\delta_1 < x \leq n\big\}$ and decrement the degrees of all groups $g_j, \ldots, g_{|L|}$ accordingly. Since degree δ_{j-1} remains unchanged, it may now match the decremented δ_j. This violation of (`I1`) is resolved by *merging* both groups. Due to (`I2`), the union of g_{j-1} and g_j contains consecutive ids and it suffices to grow $n_{j-1} \leftarrow n_{j-1}{+}n_j$ and to delete

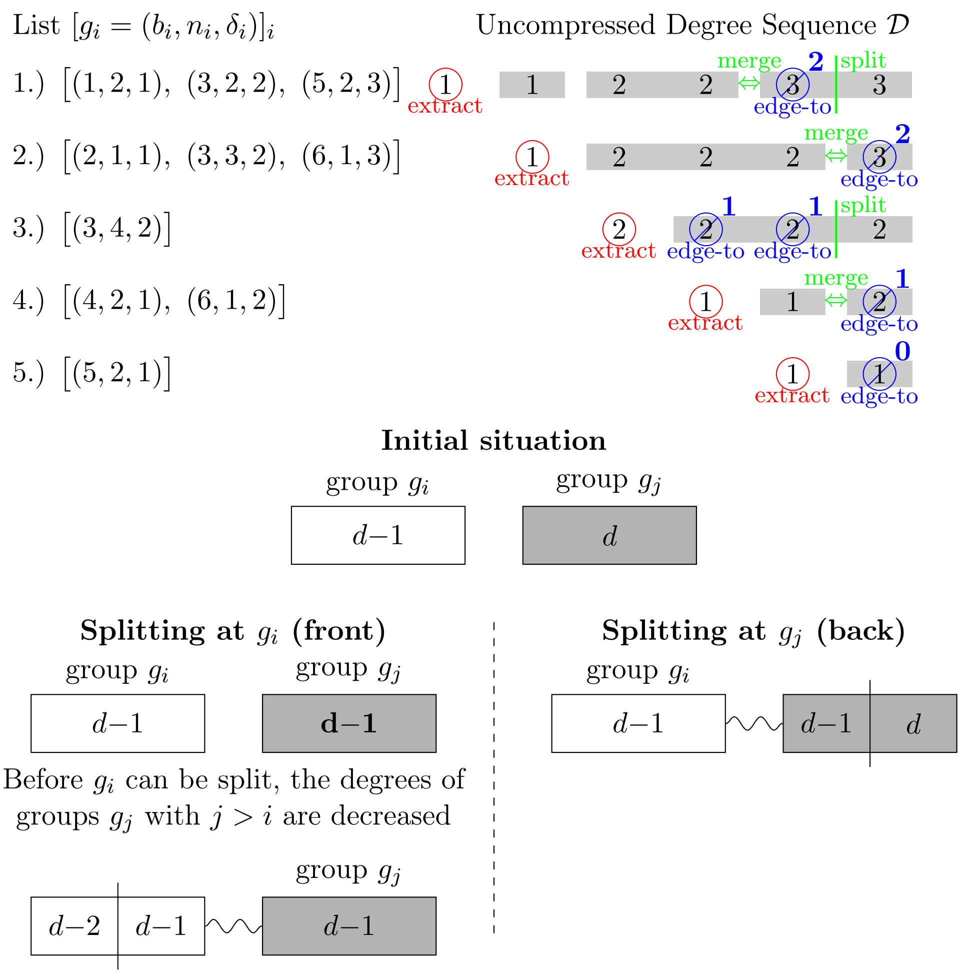


Figure 3.5: **Top:** *EM-HH* on $\mathcal{D} = (1,1,2,2,3,3)$. Values of L and $\mathcal{D}$ in row i correspond to the state at the beginning of the i-th iteration. Groups are visualized directly after extraction of the head node. The number next to an `edge-to` symbol indicates the new degree. After these updates, splitting and merging takes place. For instance, in the initial round the first node v_1 is extracted from g_1 and connected to the first node v_5 of the last group. Hence group g_3 of nodes with degree 3 is split, into node v_5 with now $\deg(v_5) = 2$ and v_6 remaining at $\deg(v_6) = 3$. Since group g_2 of nodes $\{v_3, v_4\}$ has also degree 2 it is merged with the new group of v_5. **Bottom:** Consider two adjacent groups g_i, g_j with degrees $d-1$ and d. A split of g_i (left) or g_j (right) directly triggers a merge, so the number of groups remains the same.

group g_j (see Figure 3.5 step 2 in which the degree of g_3 is reduced to $d_3 = 2$ triggering a merge with g_2).

(C2) If v_{b_1} connects only to a number $a < n_j$ of nodes in group g_j, we *split* g_j into two groups g'_j and g''_j containing nodes $[\,v_{b_j+i}\,]_{i=0}^{a-1}$ and $[\,v_{b_j+i}\,]_{i=a}^{n_j}$ respectively. We then connect node u to all a nodes in the first fragment g'_j and hence need to decrease its degree. Thus, a merge analogous to (C1) may be required if degree δ_{j-1} matches the decreased degree of group g'_j (see Figure 3.5 step 1 in which group g_3 is split into two fragments with degrees $d_{3'} = 2$ and $d_3 = 3$ respectively, triggering a merge between group g_2 and fragment $g_{3'}$). Afterwards, the degrees of groups $g_{j+1}, \ldots, g_{|L|}$ are decreased wholly as in (C1).

If the requested degree δ_1 cannot be met (i.e., $\delta_1 > \sum_{k=1}^{|L|} n_k$), the input is not graphical [Hak62]. However, a sufficiently large random powerlaw degree sequence contains at most very few nodes that cannot be materialized as requested since the vast majority of nodes have low degrees. Thus, we do not explicitly ensure that the sampled degree sequence is graphical and rather correct the negligible inconsistencies later on by ignoring the unsatisfiable requests.

3.5.3 Improving the I/O-complexity

In *EM-HH*'s current formulation, it requires $\mathcal{O}(m)$ time which is already optimal in case edges have to be emitted. Testing whether $\mathcal{D}$ is graphical however is sub-optimal. We thus introduce a simple optimization, which also yields optimality for these tests, improves constant factors and gives I/O-optimal accesses.

Observe that only groups in the vicinity of g_j can be split or merged; we call these the *active* frontier. In contrast, the so-called *stable* groups $g_{j+1}, \ldots, g_{D(\mathcal{D})}$ keep their relative degree differences as the pending degrees of all their nodes are decremented by one in each iteration. Further, they will become neighbors to all subsequently extracted nodes until group g_{j+1} eventually becomes an active merge candidate. Thus, we do not have to update the degrees of stable groups in every round, but rather maintain a single global iteration counter I and count how many iterations a group remained stable: when a group g_k becomes stable in iteration I_0, we annotate it with I_0 by adding $\delta_k \leftarrow \delta_k + I_0$. If g_k has to be activated again in iteration $I > I_0$, its updated degree follows as $\delta_k \leftarrow \delta_k - I$. The degree δ_k remains positive since (`I1`) enforces a timely activation.

Lemma 3.4. *The optimized variant of EM-HH requires* $\mathcal{O}(\text{scan}(D(\mathcal{D})))$ *I/Os if* L *is stored in an external memory list.*

Proof. An external-memory list requires $\mathcal{O}(\text{scan}(k))$ I/Os to execute any sequence of k sequential read, insertion, and deletion requests to adjacent positions (i.e. if no seeking is necessary) [Pag03]. We will argue that *EM-HH* scans L roughly twice, starting simultaneously from the front and back.

Every iteration starts by extracting a node of minimal degree. Doing so corresponds to accessing and eventually deleting the list's first element g_i. If the list's head block is

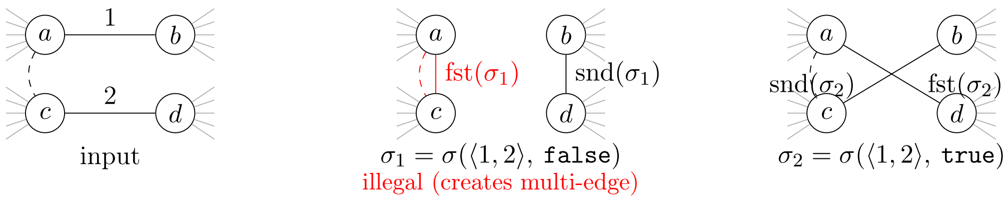

1
2
a
b
c
d
input
fst(σ_1)
snd(σ_1)
$\sigma_1 = \sigma(\langle 1,2\rangle, \texttt{false})$
illegal (creates multi-edge)
snd(σ_2)
fst(σ_2)
$\sigma_2 = \sigma(\langle 1,2\rangle, \texttt{true})$

Figure 3.6: Any exchange of exactly one node between two edges in an undirected graph (left) yields one of two isomorphic results (middle and right). We encode a swap with the two edge ids (i.e. their rank in E_L) and a direction flag selecting one of the two possible swaps. In the example, the swap σ_1 (middle) is illegal, as it introduces the edge $\{a, c\}$ which already exists.

cached, we only incur an I/O after deleting $\Theta(B)$ head groups, yielding $\mathcal{O}(\text{scan}(D(\mathcal{D})))$ I/Os during the whole execution. The same is true for accesses to the back of the list: the minimal degree increases monotonically during the algorithm's execution until the extracted node has to be connected to all remaining vertices. In a graphical sequence, this implies that only one group remains and we can ignore the simple base case asymptotically. Neglecting splitting and merging, the distance between the list's head and the *active* frontier decreases monotonically triggering $\mathcal{O}(\text{scan}(D(\mathcal{D})))$ I/Os.

Merging. As described before, it may be necessary to reactivate *stable* groups, i.e. to reload the group behind the active frontier (towards L's end). Thus, we not only keep the block F containing the frontier cached, but also block G behind it. It does not incur additional I/O, since we are scanning backwards through L and already read G before F. The reactivation of *stable* groups hence only incurs an I/O when the whole block G is consumed and deleted. Since this does not happen before $\Omega(B)$ merges take place, reactivations may trigger $\mathcal{O}(\text{scan}(D(\mathcal{D})))$ I/Os in total.

Splitting. Splitting does not influence *EM-HH*'s asymptotic I/O complexity: Only an active group of degree d can be split yielding two fragments of degrees $d-1$ and d respectively. A second split of one of these fragments does not increase the number of groups since two of the three involved fragments have to be merged (cf. Figure 3.5). As a result splitting can at most double L's size. □

3.6 *EM-ES*: I/O-efficient Edge Switching

EM-ES implements an external memory edge switching algorithm to randomize networks. Following *LFR*'s original usage of FDSM, *EM-ES* is crucial in *EM-LFR* to randomize the inter-community graph as well as all communities independently (cf. Figure 3.3), and additionally functions as a building block to rewire illegal edges (cf sections 3.7 and 3.8). As discussed in section 3.10.6, the algorithm also has applications as a standalone tool in network analysis.

EM-ES applies a sequence $S = [\,\sigma_s\,]_{s=1}^{k}$ of edge swaps σ_s to a simple graph $G = (V, E)$, where the parameter k is typically chosen as $k \in [1m, 100m]$. The graph is represented by a lexicographically ordered edge list $E_L = [\, e_i \,]_{i=1}^{m}$ which contains for every ordered edge $[u, v] \in E$ (i.e. $u < v$) only the entry (u, v) and omits (v, u). We encode a swap $\sigma(\langle a, b\rangle,\ d)$ as a three-tuple with a direction bit d and the two indices a, b of the edges $e_a, e_b \in E_L$ that are supposed to be swapped. As illustrated in Figure 3.6, a swap simply exchanges one of the two incident nodes of each edge where d selects which one. More formally, we denote the two resulting edges as $\text{fst}(\sigma(\langle a, b\rangle,\ d))$ and $\text{snd}(\sigma(\langle a, b\rangle,\ d))$ with

$$\text{fst}(\sigma(\langle a, b\rangle,\ d)) := \begin{cases} \{\alpha_1, \beta_1\} & \text{if } d = \texttt{false} \\ \{\alpha_1, \beta_2\} & \text{if } d = \texttt{true} \end{cases}$$

$$\text{snd}(\sigma(\langle a, b\rangle,\ d)) := \begin{cases} \{\alpha_2, \beta_2\} & \text{if } d = \texttt{false} \\ \{\alpha_2, \beta_1\} & \text{if } d = \texttt{true} \end{cases},$$

where $[\alpha_1, \alpha_2] = e_a$ and $[\beta_1, \beta_2] = e_b$ are the edges at ranks a and b in the edge list E_L. In unambiguous cases, we shorten the expressions to $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$ respectively. The swap's constituents a and b are typically drawn independently and uniformly at random. Thus, the sequence can contain illegal swaps that would introduce either multi-edges or self-loops. Such illegal swaps are simply skipped. In order to do so, the following tasks have to be addressed for each $\sigma(\langle a, b\rangle,\ d)$:

(`T1`) Gather the nodes incident to edges e_a and e_b.

(`T2`) Compute $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$ and skip if a self-loop arises.

(`T3`) Verify that the graph remains simple, i.e. skip if edge $\text{fst}(\sigma)$ or $\text{snd}(\sigma)$ already exist in E_L.

(`T4`) Update the graph representation E_L.

If the whole graph fits in IM, a hash set per node storing all neighbors can be used for adjacency queries and updates in expected constant time (e.g., *VL-ES* [VL16]). Then, (`T3`) and (`T4`) can be executed for each swap in expected time $\mathcal{O}(1)$. However, in the EM model this approach incurs $\Omega(1)$ I/Os per swap with high probability for a graph with $m \geq cM$ and any constant $c > 1$.

We address this issue by processing the sequence of swaps S batchwise in chunks of size $r = \Theta(m)$ which we call *runs*. As illustrated in Figure 3.7, *EM-ES* executes several phases for each run. While they roughly correspond to the four tasks outlined above, the algorithm is more involved as it has to explicitly track data dependencies between swaps within a batch. There are two types: A *source edge dependency* occurs if (at least) two swaps share the same edge id as source. In this case, successfully executing the first swap will replace the edge by another one. This update has to be communicated to all later swaps involving this edge id. *Target edge dependencies* exist because swaps must not introduce multi-edges. Therefore, each swap has to assert that none of its new edges (target edges) are already present in the graph. For this reason, *EM-ES* has to inform a swap about the creation or deletion of target edges that occurred earlier in the run.

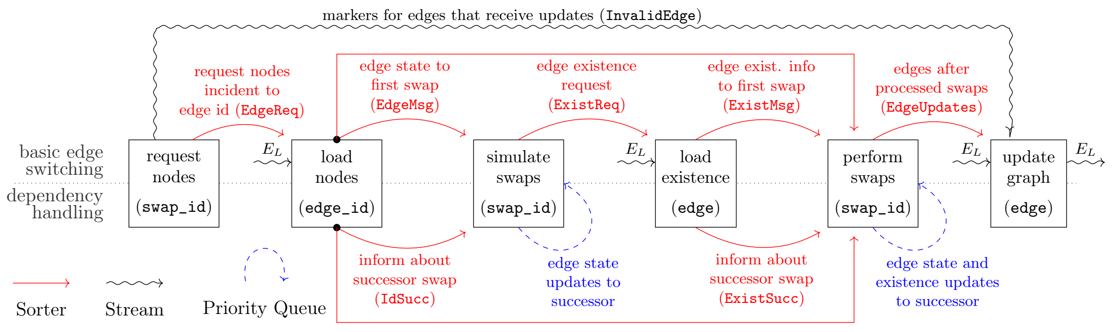

markers for edges that receive updates (InvalidEdge)
request nodes incident to edge id (EdgeReq)
edge state to first swap (EdgeMsg)
edge existence request (ExistReq)
edge exist. info to first swap (ExistMsg)
edges after processed swaps (EdgeUpdates)
basic edge switching
dependency handling
request nodes (swap_id)
E_L
load nodes (edge_id)
simulate swaps (swap_id)
E_L
load existence (edge)
perform swaps (swap_id)
E_L
update graph (edge)
E_L
inform about successor swap (IdSucc)
edge state updates to successor
inform about successor swap (ExistSucc)
edge state and existence updates to successor
Sorter
Stream
Priority Queue

Figure 3.7: Data flow during an *EM-ES* run. Communication between phases is implemented via EM sorters, self-loops use a PQ-based *TFP*. Brackets within a phase represent the type of elements that are iterated over. If multiple input streams are used, they are joined with this key. Independent swaps as in section 3.6.1 require only communication via sorters as shown on the upper half.

3.6.1 *EM-ES* for Independent Swaps

For simplicity's sake, we first assume that all swaps are independent, i.e. that there are neither *source edge* nor *target edge* dependencies in a run. Section 3.6.2 contains the algorithmic modifications necessary to account for dependencies.

The design of *EM-ES* is driven by the intuition that there are three types of cross-referenced data, namely (i) the sequence of swaps ranked in the order they were issued, (ii) edges addressed by their indices (e.g., to load and store their incident nodes) and (iii) edges referenced by their constituents (in order to query their existence). To resolve these unstructured references, the algorithm is decomposed into several phases and iterates in each phase over one of these data types in order. There is no pipelining, so a new phase only starts processing when the previous is completed. Similarly to Time-Forward Processing (cf. section 3.2.3), phases communicate by sending messages addressed to the key of the receiving phase. The messages are pushed into a sorter[7] to later be processed in the order dictated by the data source of the receiving end. *EM-ES* uses the following phases:

Request nodes and *load nodes*

The goal of these two phases is to load the constituents of the edges referenced by the run's swaps. We iterate over the sequence S of swaps. For the s-th swap $\sigma(\langle a, b\rangle, d)$, we push two messages `edge_req`$(a, s, 0)$ and `edge_req`$(b, s, 1)$ into the sorter `EdgeReq`. A

[7]The term *sorter* refers to a container with two modes of operation: in the first phase, items are pushed into the write-only sorter in an arbitrary order by some algorithm. After an explicit switch, the filled data structure becomes read-only and the elements are provided as a lexicographically non-decreasing stream which can be rewound at any time. While a sorter is functionally equivalent to filling, sorting and reading back an EM vector, the restricted access model reduces constant factors in the implementation's runtime and I/O-complexity [BDS09].

message's third entry encodes whether the request is issued for the first or second edge of a swap. This information only becomes relevant when we allow dependencies. *EM-ES* then scans in parallel through the edge list E_L and the requests `EdgeReq`, which are now sorted by edge ids. If there is a request `edge_req`(i, s, p) for an edge $e_i = [u, v]$, the edge's node pair is sent to the requesting swap by pushing a message `edge_msg`$(s, p, (u, v))$ into the sorter `EdgeMsg`.

Additionally, for every edge we push a bit into the sequence `InvalidEdge`, which is asserted iff an edge received a request. These edges are considered invalid and will be deleted when updating the graph in section 3.6.1. Since both phases produce only a constant amount of data per input element, we obtain an I/O complexity of $\mathcal{O}(\text{sort}(r) + \text{scan}(m))$.

Simulate swaps and *load existence*

The two phases gather all information required to decide whether a swap is legal. *EM-ES* scans through the sequence S of swaps and `EdgeMsg` in parallel: For the s-th swap $\sigma(\langle a, b\rangle,\ d)$, there are exactly two messages `edge_msg`$(s, 0, e_a)$ and `edge_msg`$(s, 1, e_b)$ in `EdgeMsg`. This information suffices to compute the switched edges $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$, but not to test for multi-edges.

It remains to check whether the switched edges already exist; we push the existence requests `exist_req`$(\text{fst}(\sigma), s)$ and `exist_req`$(\text{snd}(\sigma), s)$ into the sorter `ExistReq`. Observe that for *request nodes* we use the node pairs rather than edge ids, which are not well defined here. Afterwards, a parallel scan through the edge list E_L and `ExistReq` is performed to answer the requests. Only if an edge e requested by swap id s is found, the message `exist_msg`(s, e) is pushed into the sorter `ExistMsg`. Both phases hence incur a total of $\mathcal{O}(\text{sort}(r) + \text{scan}(m))$ I/Os.

Perform swaps

We rewind the `EdgeMsg` sorter and jointly scan through the sequence of swaps S and the sorters `EdgeMsg` and `ExistMsg`. As described in the simulation phase, *EM-ES* computes the switched edges $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$ from the original state e_a and e_b. The swap is considered illegal if a switched edge is a self-loop or if an existence info is received via `ExistMsg`. If σ is legal we push the switched edges $\text{fst}(\sigma)$ and $\text{snd}(\sigma)$ into the sorter `EdgeUpdates`, otherwise we propagate the unaltered source edges e_a and e_b. This phase requires $\mathcal{O}(\text{sort}(r))$ I/Os.

Update edge list

The new edge list E'_L is obtained by merging the original lexicographic increasing list E_L and the sorted updated edges `EdgeUpdates`, triggering $\mathcal{O}(\text{scan}(m))$ I/Os. During this process, we skip all edges in E_L that are flagged invalid in the bit stream `InvalidEdge`. The result is a sorted new E'_L with $|E'_L| = m$ edges that can be fed into the next run.

3.6.2 Inter-Swap Dependencies

In this section, we introduce the modifications necessary due to dependencies between swaps within a run. In its final version, *EM-ES* produces the same result as a sequential processing of S.

Source edge dependencies are detected during the *load nodes* phase since multiple requests for the same edge id arrive. We record these dependencies as an explicit dependency chain along which intermediate updates can be propagated. Target edge dependencies surface in the *load existence* phase since multiple existence requests and notifications arrive for the same edge. Again, an explicit dependency chain is computed. During the *perform swaps* phase, *EM-ES* forwards the source edge states and existence updates to successor swaps using information from both dependency chains.

Target edge dependencies

Consider the case where a swap $\sigma_{s_1}(\langle a, b\rangle,\ d)$ changes the state of edges e_a and e_b to $\mathrm{fst}(\sigma_1)$ and $\mathrm{snd}(\sigma_1)$ respectively. Later, a second swap σ_2 inquires about the existence of either of the four edges which has obviously changed compared to the initial state. We extend the simulation phase to track such edge modifications and not only push messages `exist_req`$(\mathrm{fst}(\sigma_1), s_1)$ and `exist_req`$(\mathrm{snd}(\sigma_1), s_1)$ into sorter `EdgeReq`, but also report that the original edges may change (during simulation phase it is unknown whether the swap has to be skipped). This is implemented by pushing the messages `exist_req`$(e_a, s_1,$ `may_change`$)$ and `exist_req`$(e_b, s_1,$ `may_change`$)$ into the same sorter.

In case of dependencies, multiple messages are received for the same edge e during the *load existence* phase. If so, only the request of the first swap involved is answered as before. Also, every swap σ_{s_1} is informed about its direct successor σ_{s_2} (if any) by pushing the message `exist_succ`(s_1, e, s_2) into the sorter `ExistSucc`, yielding the aforementioned dependency chain. As an optimization, `may_change` requests at the end of a chain are discarded since no recipient exists.

During the *perform swaps* phase, *EM-ES* executes the same steps as described earlier. The swap may receive a successor for every edge it sent an existence request to, and informs each successor about the state of the appropriate edge after the swap is processed.

Source edge dependencies

Consider two swaps $\sigma_{s_1}(\langle a_1, b_1\rangle,\ d_1)$ and $\sigma_{s_2}(\langle a_2, b_2\rangle,\ d_2)$ with $s_1{<}s_2$ which share a source edge id, i.e. $\{a_1, b_1\} \cap \{a_2, b_2\}$ is non-empty. This dependency is detected during the *load nodes* phase since requests `edge_req`(e_i, s_1, p_1) and `edge_req`(e_i, s_2, p_2) arrive for edge id e_i. In this case, we answer only the request of s_1 and build a dependency chain as before using messages `id_succ`(s_1, p_1, s_2, p_2) pushed into the sorter `IdSucc`.

During the simulation phase, *EM-ES* cannot yet decide whether a swap is legal. Thus, s_1 sends for every conflicting edge its original state as well as the updated state to the p_2-th slot of s_2 using a PQ. If a swap receives multiple edge states per slot, it simulates the swap for all possible combinations.

During the *perform swaps* phase, *EM-ES* operates as described in the independent case: it computes the swapped edges and determines whether the swap has to be skipped. If a successor exists, the new state is not pushed into the `EdgeUpdates` sorter but rather forwarded to the successor in a *TFP* fashion. This way, every invalidated edge id receives exactly one update in `EdgeUpdates` and the merging remains correct.

3.6.3 Complexity

Due to source edge dependencies, *EM-ES*'s complexity increases with the number of swaps that share the same edge id. This number is low in case $r = \mathcal{O}(m)$: let X_i be a random variable expressing the number of swaps that reference edge e_i. Since every swap constitutes two independent Bernoulli trials towards e_i, the indicator X_i is binomially distributed with $p = 1/m$, yielding an expected chain length of $2r/m$. Also, for $r = m/2$ swaps, $\max_{1 \leq i \leq n}(X_i) = \mathcal{O}(\ln(m)/\ln\ln(m))$ holds with high probability based on a balls-into-bins argument [MR95]. Thus, we can bound the largest number of edge states simulated with high probability by $\mathcal{O}(\text{polylog}(m))$, assuming non-overlapping dependency chains. Further, observe that X_i converges towards an independent Poisson distribution for large m. Then the expected state space per edge is $\mathcal{O}(1)$. The experiments in section 3.10.3 suggest that this bound also holds for overlapping dependency chains.

In order to keep the dependency chains short, *EM-ES* splits the sequence of swaps S into runs of equal size. Our experimental results show that a run size of $r = m/8$ is a suitable choice. For every run, the algorithm executes the six phases as described before. Each time the graph is updated, the mapping between an edge and its id may change. The switching probabilities, however, remain unaltered due to the initial assumption of uniformly distributed swaps. Thus, *EM-ES* triggers $\mathcal{O}(k/m \, \text{sort}(m))$ I/Os in total with high probability.

3.7 *EM-CM/ES*: Sampling of random graphs from prescribed degree sequence

In this section, we propose an alternative approach to generate a graph from a prescribed degree sequence. In contrast to *EM-HH* which generates a highly biased but simple graph, we use the Configuration Model to sample a random but in general non-simple graph. Thus, the resulting graph may contain self-loops and multi-edges which we then rewire to obtain a simple graph. As experimental data suggests (cf. section 3.7.2), this still results in a biased realization of the degree sequence requiring additional edge switching randomization steps.

3.7.1 Configuration Model

Let $\mathcal{D} = [\, d_i \,]_{i=1}^{n}$ be a degree sequence with n nodes. The Configuration Model builds a multiset of node ids which can be thought of as *half-edges* (or stubs). It produces a total of d_i *half-edges* labeled v_i for each node v_i. The algorithm then chooses two half-edges

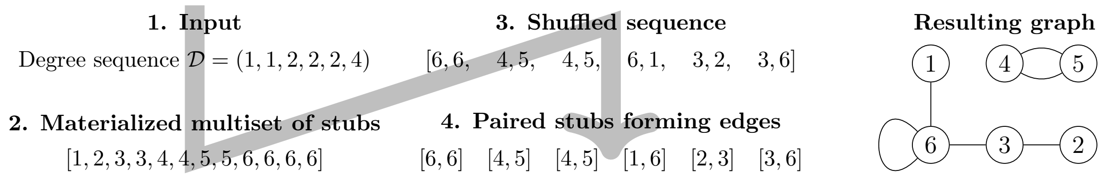

1. Input
Degree sequence $\mathcal{D} = (1, 1, 2, 2, 2, 4)$
2. Materialized multiset of stubs
$[1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6]$
3. Shuffled sequence
$[6, 6, \quad 4, 5, \quad 4, 5, \quad 6, 1, \quad 3, 2, \quad 3, 6]$
4. Paired stubs forming edges
$[6, 6] \quad [4, 5] \quad [4, 5] \quad [1, 6] \quad [2, 3] \quad [3, 6]$
Resulting graph
1
4
5
6
3
2

Figure 3.8: A Configuration Model run on degree sequence $\mathcal{D} = (1, 1, 2, 2, 2, 4)$.

uniformly at random and creates an edge according to their labels. It repeats the last step with the remaining half-edges until all are paired. A naïve implementation of this algorithm requires with high probability $\Omega(m)$ I/Os if $m \geq cM$ and any constant $c > 1$. It is therefore impractical in the fully external setting.

We rather materialize the multiset as a sequence in which each node appears d_i times similar to the approach of [KWZ15]. Subsequently, the sequence is shuffled to obtain a random permutation with $\mathcal{O}(\text{sort}(m))$ I/Os by sorting the sequence according to a uniform variate drawn for each half-edge[8]. Finally, we scan over the shuffled sequence and match pairs of adjacent half-edges into edges.

As illustrated in Figure 3.8, the Configuration Model gives rise to self-loops and multi-edges which then need to be rewired, cf. section 3.7.2. Consequently, the rewiring process depends on the number of introduced illegal edges. In the following lemma, we bound their number from above.

Lemma 3.5. *Let $\mathcal{D}$ be drawn from* $\textsc{Pld}\left([a, b), 2\right)$. *The expected number of self-loops and multi-edges are bound by*

$$\mathbb{E}[\#\textit{self-loops}] \leq \frac{1}{2}\left(\frac{b - a + 1}{\ln(b + 1) - \ln(a)}\right) \textit{ and } \mathbb{E}[\#\textit{multi-edges}] \leq \frac{1}{2}\left(\frac{b - a + 1}{\ln(b + 1) - \ln(a)}\right)^2.$$

Proof. [AHH19] and [New10] derive the expectation values for an arbitrary degree sequence $\mathcal{D}$ in terms of its mean $\langle\mathcal{D}\rangle$ and second moment $\langle\mathcal{D}^2\rangle$. In the limit of $n \to \infty$, the authors show

$$\mathbb{E}[\#\text{self-loops}(\mathcal{D})] = \frac{\langle\mathcal{D}^2\rangle - \langle\mathcal{D}\rangle}{2(\langle\mathcal{D}\rangle - 1/n)} \longrightarrow \frac{\langle\mathcal{D}^2\rangle - \langle\mathcal{D}\rangle}{2\langle\mathcal{D}\rangle} \tag{3.1}$$

$$\mathbb{E}[\#\text{multi-edges}(\mathcal{D})] \leq \frac{1}{2}\left(\frac{(\langle\mathcal{D}^2\rangle - \langle\mathcal{D}\rangle)^2}{(\mathcal{D} - 1/n)(\mathcal{D} - 3/n)}\right) \longrightarrow \frac{1}{2}\left(\frac{\langle\mathcal{D}^2\rangle - \langle\mathcal{D}\rangle}{\langle\mathcal{D}\rangle}\right)^2. \tag{3.2}$$

We now bound $\langle\mathcal{D}\rangle$ and $\langle\mathcal{D}^2\rangle$ in the case that $\mathcal{D}$ is drawn from the powerlaw distribution $\textsc{Pld}\left([a, b), \gamma\right)$. Since each entry in $\mathcal{D}$ is independently drawn, it suffices to bound the expected value and the second moment of the underlying distribution. Then, they are given by $\langle\mathcal{D}\rangle = (\sum_{i=a}^{b} i^{-\gamma+1})/C_\mathcal{D}$ and $\langle\mathcal{D}^2\rangle = (\sum_{i=a}^{b} i^{-\gamma+2})/C_\mathcal{D}$ where $C_\mathcal{D} = \sum_{i=a}^{b} i^{-\gamma}$. Both

[8]If $M > \sqrt{mB}(1 + o(1)) + \mathcal{O}(B)$ this can be improved to $\mathcal{O}(\text{scan}(m))$ I/Os [San98] which does however not affect the total complexity of our pipeline.

numerators are sandwiched between the two integrals $\int_a^{b+1} x^q \,\mathrm{d}x \leq \sum_{i=a}^{b} i^q \leq \int_{a-1}^{b} x^q \,\mathrm{d}x$ where $q = -\gamma{+}1$ or $q = -\gamma{+}2$ respectively. In the case of $\gamma = 2$, the second moment hence simplified to $\langle \mathcal{D}^2 \rangle = (\sum_{i=a}^{b} 1)/C_{\mathcal{D}} = (b{-}a{+}1)/C_{\mathcal{D}}$. Applying this identity and the lower bound $(\int_a^{b+1} x^{-1} \,\mathrm{d}x)/C_{\mathcal{D}} \leq \langle \mathcal{D} \rangle$ to Eqs. 3.1 and 3.2, directly yields the claim. $\square$

3.7.2 Edge rewiring for non-simple graphs

Graphs generated using the Configuration Model may contain multi-edges and self-loops. In order to obtain a simple graph we need to detect these illegal edges and rewire them. After sorting the edge list lexicographically, illegal edges can be detected in a single scan. For each self-loop we issue a swap with a randomly selected partner edge. Similarly, for each group of parallel edges, we generate swaps with random partner edges for all but one multi-edge. Subsequently, we execute the provisioned swaps using a variant of *EM-ES* (see below). The process is repeated until all illegal edges have been removed. To accelerate the endgame, we double the number of swaps for each remaining illegal edge in every iteration.

Since *EM-ES* is employed to remove parallel edges based on targeted swaps, it needs to process non-simple graphs. Analogous to the initial formulation, we forbid swaps that introduce multi-edges even if they would reduce the multiplicity of another edge (cf. [Zha13]). Nevertheless, *EM-ES* requires slight modifications for non-simple graphs.

Consider the case where the existence of a multi-edge is inquired several times. Since E_L is sorted, the initial edge multiplicities can be counted while scanning E_L during the *load existence* phase. In order to correctly process the dependency chain, we have to forward the (possibly updated) multiplicity information to successor swaps. We annotate the existence tokens `exist_msg`$(s, e, \#(e))$ with these counters where $\#(e)$ is the multiplicity of edge e.

More precisely, during the *perform swaps* phase, swap $\sigma_1 = \sigma(\langle a, b\rangle,\ d)$ is informed (among others) of multiplicities of edges $e_a, e_b, \mathrm{fst}(\sigma_1)$ and $\mathrm{snd}(\sigma_1)$ by incoming existence messages. If σ_1 is legal, we send requested edges and multiplicities of the swapped state to any successor σ_2 of σ_1 provided in `ExistSucc`. Otherwise, we forward the edges and multiplicities of the unchanged initial state. As an optimization, edges which have been removed (i.e. have multiplicity zero) are omitted.

3.8 *EM-GER*/*EM-CER*: Merging and repairing the intra- and inter-community graphs

As illustrated in Figure 3.3, *LFR* samples the inter-community graph and all intra-community graphs independently. As a result, they may exhibit minor inconsistencies which *EM-LFR* resolves in accordance with the original version by applying additional rewiring steps which are discussed in this section.

3.8.1 *EM-GER*: Global Edge Rewiring

The global graph is materialized without taking the community structure into account. As illustrated in Figure 3.2 (center), it therefore can contain edges between nodes that share a community. Those edges have to be removed as they decrease the mixing parameter μ. We rewire these edges by performing an edge swap for each forbidden edge with a randomly selected partner. Since it is unlikely that such a random swap introduces another illegal edge (if sufficiently many communities exist), this probabilistic approach effectively removes forbidden edges. We apply this idea iteratively and perform multiple rounds until no forbidden edges remain.

To detect illegal edges, *EM-GER* considers the community assignment's output which is a lexicographically ordered sequence χ of (v, ξ)-pairs containing the community ξ for each node v. For nodes that join multiple communities several such pairs exist. Based on this, we annotate every edge with the communities of both incident vertices by scanning through the edge list twice: once sorted by source nodes and once by target nodes. For each forbidden edge, a swap is generated by drawing a random partner edge id and a swap direction. Subsequently, all swaps are executed using *EM-ES* which now also emits the set of edges involved. It suffices to restrict the scan for illegal edges to this set since all edges not contained are legal by construction.

Complexity. Each round requires $\mathcal{O}(\text{sort}(m))$ I/Os for selecting the edges and executing the swaps. The number of rounds is usually small but depends on the community size distribution: the probability that a randomly placed edge lies within a community increases with the size of the community.

3.8.2 *EM-CER*: Community Edge Rewiring

In the case of overlapping communities, the same edge can be generated as part of multiple communities. We iteratively apply semi-random swaps to remove those parallel edges similarly to sections 3.7.2 and 3.8. The selection of random partners is however more involved for *EM-CER* as it has to ensure that all swaps take place between two edges of the same community. This way, the rewired edges keep the same memberships as their sources and the community sizes do not change. The rewiring itself is easy to achieve by considering all communities independently.

Unfortunately, *EM-CER* needs to process all communities conjointly to detect forbidden edges: we augment each edge $[u_i, v_i]$ with its community id c_i and concatenate these lists into one annotated graph possibly containing multi-edges. During a scan through the lexicographically sorted and annotated edge list $[\,(u_i, v_i, c_i)\,]_i$, parallel edges are easily found as they appear next to each other. We select all but one from each group for rewiring. Each partner is selected by a uniform edge id e_b addressing the e_b-th edge of the community at hand. In a fully external setting, it suffices to sort the selected candidates, their partners and the edge list by community to gather all information required to invoke *EM-ES*.

EM-CER avoids the expensive step of sorting all edges if we can store $\mathcal{O}(1)$ items per illegal edge in IM (which is almost certainly the case since there are typically few illegal edges). It then sorts the edge ids of partners for every community independently and keeps pointers to the smallest requested partner edge id of each community. While scanning through the concatenated edge list, we count for each community the number of edges seen so far. When the counter matches the smallest requested id of the current edge's community, we load the edge and advance the pointer to the next request.

Complexity. The fully external rewiring requires $\mathcal{O}(\text{sort}(m))$ I/Os for the initial step and each following round. The semi-external variant triggers only $\mathcal{O}(\text{scan}(m))$ I/Os per round. The number of rounds is usually small and the overall runtime spent on this step is insignificant. Nevertheless, the described scheme is a Las Vegas algorithm and there exist (unlikely) instances on which it will fail.[9] To mitigate this issue, we allow a small fraction of edges (e.g., 10^{-3}) to be removed if we detect a slow convergence. To speed up the endgame, we also draw additional swaps uniformly at random from communities which contain a multi-edge.

3.9 Implementation

We implemented the proposed algorithms in C++ based on the STXXL library [DKS08], providing implementations of EM data structures, a parallel EM sorter, and an EM priority queue. Among others, we applied the following optimizations for *EM-ES*:

- Most message types contain both a swap id and a flag indicating which of the swap's edges is targeted. We encode both of them in a single integer by using all but the least significant bit for the swap id and store the flag in there. This significantly reduces the memory volume and yields a simpler comparison operator since the standard integer comparison already ensures the correct lexicographic order.

- Instead of storing and reading the sequence of swaps several times, we exploit the implementation's pipeline structure and directly issue edge id requests for every arriving swap. Since this is the only time edge ids are read from a swap, only the remaining direction flag is stored in an efficient EM vector, which uses one bit per flag and supports I/O-efficient writing and reading. Both steps can be overlapped with an ongoing *EM-ES* run.

- Instead of storing each edge in the sorted external edge list as a pair of nodes, we only store each source node once and then list all targets of that node. This still supports sequential scan and merge operations which are the only operations we need. This almost halves the I/O volume of scanning or updating the edge list.

- During the execution of several runs we can delay the updating of the edge list and combine it with the *load nodes* phase of the next run. This reduces the number of scans per additional run from three to two.

[9]Consider a node which is a member of two communities in which it is connected to all other nodes. If only one of its neighbors also appears in both communities, the multi-edge cannot be rewired.

- We use asynchronous stream adapters for tasks such as streaming from sorters or the generation of random numbers. These adapters run in parallel in the background to preprocess and buffer portions of the stream in advance and hand them over to the main thread.

Besides parallel sorting and asynchronous pipeline stages, the current *EM-LFR* implementation facilitates parallelism during the generation and randomization of intra-community graphs which can be computed without any synchronization. While the algorithms themselves are sequential, this pipelining and parallelization of independent tasks within *EM-LFR* leads to a consistent utilization of available threads in our test system (cf. section 3.10).

3.10 Experimental Results

3.10.1 Notation and Setup

The number of repetitions per data point (with different random seeds) is denoted with S. Errorbars correspond to the unbiased estimation of the standard deviation. For *LFR* we perform experiments based on two different scenarios:

lin The maximal degrees and community sizes scale linearly as a function of n. For a particular n and ν the parameters are chosen as: $\mu \in \{0.2, 0.4, 0.6\}$, $d_{\min}{=}10\nu$, $d_{\max}{=}n\nu/20$, $\gamma{=}2$, $s_{\min}{=}20$, $s_{\max}{=}n/10$, $\beta{=}1$, $O{=}n$.

const We keep the community sizes and the degrees constant and consider only non-overlapping communities. The parameters are chosen as: $d_{\min}{=}50$, $d_{\max}{=}10\,000$, $\gamma{=}2$, $s_{\min}{=}50$, $s_{\max}{=}12\,000$, $\beta{=}1$, $O{=}n$.

Real-world networks have been shown to have increasing average degrees as they become larger [LKF05]. Increasing the maximum degree as in our first setting **lin** increases the average degree. Having a maximum community size of $n/10$ means, however, that a significant proportion of the nodes belongs to huge communities which are not very tightly knit due to the large number of nodes of low degree. While a more limited growth is probably more realistic, the exact parameters depend on the network model.

Our second parameter set **const** shows an example of much smaller maximum degrees and community sizes. We chose the parameters such that they approximate the degree distribution of the Facebook network in May 2011 when it consisted of 721 million active users as reported in [Uga+11]. The same study however found that strict powerlaw models are unable to accurately mimic Facebook's degree distribution. Further, the authors show that the degree distribution of the U.S. users (removing connections to non-U.S. users) is very similar to the one of the Facebook users of the whole world, supporting our use of just one parameter set for different graph sizes.

The minimum degree of the Facebook network is 1, but such small degrees are significantly less prevalent than a power law degree sequence would suggest, which is why we chose a value of 50. Our maximum degree of 10 000 is larger than the one reported

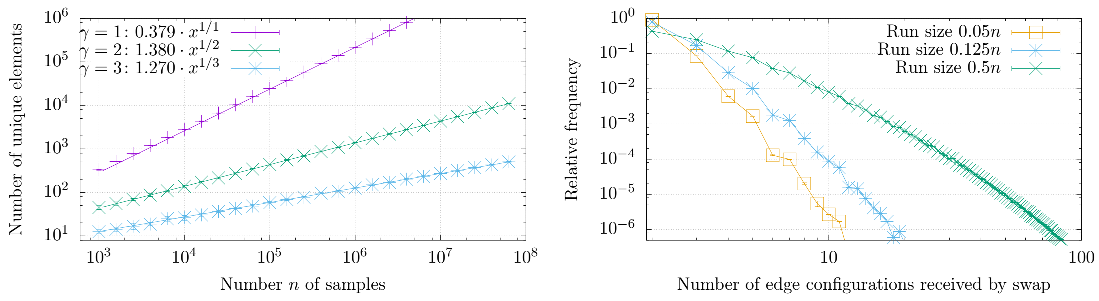


Figure 3.9: **Left:** Number of distinct elements in n samples (i.e. node degrees in a degree sequence) taken from $\textsc{Pld}([1, n), \gamma)$; cf. section 3.10.2. **Right:** Overhead induced by tracing inter-swap dependencies. Fraction of swaps as function of the number of edge configurations they receive during the simulation phase (cf. section 3.10.3).

for Facebook (5000 which is an arbitrarily enforced limit by Facebook). The expected average degree of this degree sequence is 264, which is slightly higher than the reported 190 (world) or 214 (U.S. only). Our parameters are chosen such that the median degree is approximately 99 matching the worldwide Facebook network. Similar to the first parameter set, we chose the maximum community size slightly larger than the maximum degree.

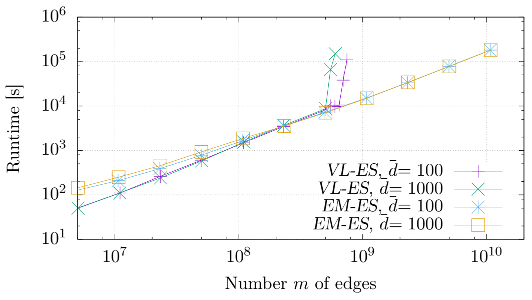
3.10.2 *EM-HH*'s state size

In Lemma 3.2, we bound *EM-HH*'s internal memory consumption by showing that a sequence of n numbers randomly sampled from $\textsc{Pld}([1, n), \gamma)$ contains only $\mathcal{O}(n^{1/\gamma})$ distinct values with high probability.

In order to support Lemma 3.2 and to estimate the hidden constants, samples of varying size between 10^3 and 10^8 are taken from distributions with exponents $\gamma \in \{1, 2, 3\}$. Each time, the number of unique elements is computed and averaged over $S = 9$ runs with identical configurations but different random seeds. The results illustrated in Figure 3.9 support the predictions with small constants and negligible deviations. For the commonly used exponent 2, we find $1.38\sqrt{n}$ distinct elements in a sequence of length n.

3.10.3 Inter-Swap Dependencies

Whenever multiple swaps target the same edge, *EM-ES* simulates all possible states to be able to retrieve conflicting edges. In section 3.6.3, we argue that the number of dependencies and the state size remains manageable if the sequence of swaps is split into sufficiently short runs. We found that for m edges and k swaps, $8k/m$ runs minimize the runtime for large instances of **lin**. As indicated in Figure 3.9, in this setting 78.7 % of swaps receive the two requested edge configurations with no additional overhead during the simulation phase. Less than 0.4 % consider more than four additional states (i.e.



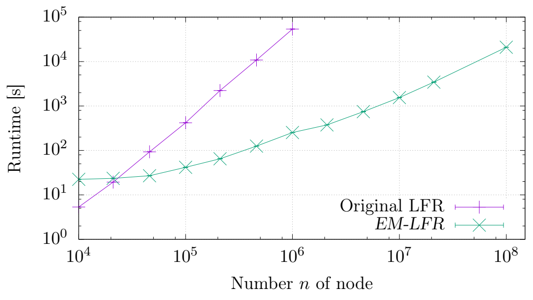


Figure 3.10: **Left:** Runtime on SysB of IM *VL-ES* and *EM-ES* on a graph with m of edges and average degree $\bar{d}$ executing $k = 10m$ swaps (cf. section 3.10.6). **Right:** Runtime on SysA of the original *LFR* implementation and *EM-LFR* for $\mu = 0.2$ (cf. section 3.10.9).

more than six messages in total). Similarly, 78.6 % of existence requests remain without dependencies.

3.10.4 Test systems

Runtime measurements were conducted on the following systems:

SysA *inexpensive compute server*: Intel Xeon E5-2630 v3 (8 cores/16 threads, 2.40GHz), 64 GB RAM, 3× Samsung 850 PRO SATA SSD (1 TB).

SysB *commodity hardware*: Intel Core i7 CPU 970 (6 cores/12 threads, 3.2GHz), 12 GB RAM, 1× Samsung 850 PRO SATA SSD (1 TB).

Since edge switching scales linearly in the number of swaps (in case of *EM-ES* in the number of runs), some of the measurements beyond 3 h runtime are extrapolated from the progress until then. We verified that errors stay within the indicated margin using reference measurements without extrapolation.

3.10.5 Performance of *EM-HH*

Our implementation of *EM-HH* produces 180 ± 5 million edges per second on SysA up to at least 2×10^{10} edges. Here, we include the computation of the input degree sequence, *EM-HH*'s compaction step, as well as the writing of the output to external memory.

3.10.6 Performance of *EM-ES*

Figure 3.10 presents the runtime required on SysB to process $k = 10m$ swaps in an input graph with m edges and for the average degrees $\bar{d} \in \{100, 1000\}$. For reference, we include the performance of the existing internal memory edge swap algorithm *VL-ES* based on the authors' implementation [VL16].[10] *VL-ES* slows down by a factor of 25 if the data structure exceeds the available internal memory by less than 10 %. We observe an analogous behavior on machines with larger RAM. *EM-ES* is faster than *VL-ES* for all instances with $m > 2.5 \times 10^8$ edges; those graphs still fit into main memory.

FDSM has applications beyond synthetic graphs, and is for instance used on real data to assess the statistical significance of observations [SHZ15]. In that spirit, we execute *EM-ES* on an undirected version of the crawled ClueWeb12 graph's core [The13] which we obtain by deleting all nodes corresponding to uncrawled URLs.[11] Performing $k = m$ swaps on this graph with $n \approx 9.8 \times 10^8$ nodes and $m \approx 3.7 \times 10^{10}$ edges is feasible in less than 19.1 h on SysB.

Bhuiyan et al. propose a distributed edge switching algorithm and evaluate it on a compute cluster with 64 nodes each equipped with two Intel Xeon E5-2670 2.60GHz 8-core processors and 64GB RAM [Bhu+14]. The authors report to perform $k = 1.15 \times 10^{11}$ swaps on a graph with $m = 10^{10}$ generated in a preferential attachment process in less than 3 h. We generate a preferential attachment graph using an EM generator [MP16] matching the aforementioned properties and carried out edge swaps using *EM-ES* on SysA. We observe a slow down of only 8.3 on a machine with 1/128 the number of comparable cores and 1/64 of internal memory.

3.10.7 Performance of *EM-CM/ES* and qualitative comparison with *EM-ES*

In section 3.7, we describe an alternative graph sampling method. Instead of seeding *EM-ES* with a highly biased graph using *EM-HH*, we employ the Configuration Model to generate a non-simple random graph and then obtain a simple graph using several *EM-ES* runs in a Las-Vegas fashion.

Since *EM-ES* scans through the edge list in each iteration, runs with very few swaps are inefficient. For this reason, we start the subsequent Markov chain to further randomize the graph early: First identify all multi-edges and self-loops and generate swaps with random partners. In a second step, we then introduce additional random swaps until the run contains at least $m/10$ operations.[12]

For an experimental comparison between *EM-ES* and *EM-CM/ES*, we consider the runtime until both yield a sufficiently uniform random sample. Of course, the uniformity is hard to quantify; similarly to related studies (cf. section 3.1.1), we estimate the mixing times of both approaches as follows. Starting from a common seed graph $G^{(0)}$, we generate an ensemble $\{G_1^{(k)}, \ldots, G_S^{(k)}\}$ of $S \gg 1$ instances by applying independent random sequences of $k \gg m$ swaps each. During this process, we regularly export

[10]Here we report only on the edge swapping process excluding any precomputation. To achieve comparability, we removed connectivity tests, fixed memory management issues, and adopted the number of swaps. Further, we extended counters for edge ids and accumulated degrees to 64 bit integers in order to support experiments with more than 2^{30} edges.

[11]We consider such vertices atypically simple as they have degree 1 and account for ≈84 % of nodes in the original graph.

[12]We chose this number as it yields execution times similar to the $m/8$-setting of *EM-ES* on simple graphs.

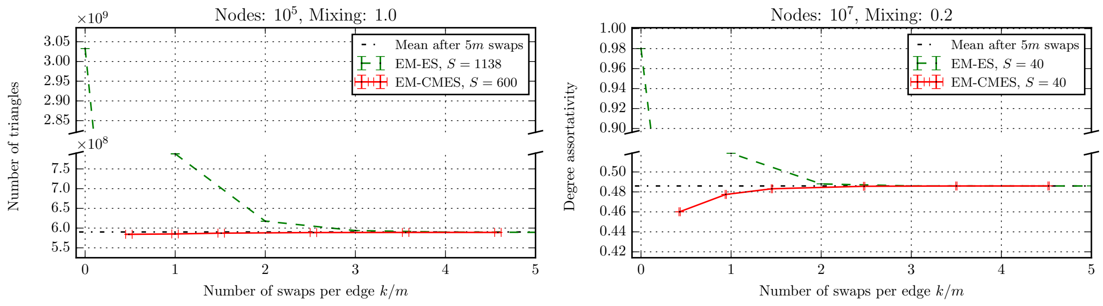


Figure 3.11: **Left:** Number of triangles on **const** with $n = 1 \times 10^5$ and $\mu = 1.0$. **Right:** Degree assortativity on **const** with $n = 1 \times 10^7$ and $\mu = 0.2$. In order to factor in the increased runtime of *EM-CM/ES* compared to *EM-HH*, plots of *EM-CM/ES* are shifted by the runtime of this phase relative to the execution of *EM-ES*. As *EM-CM/ES* is a Las-Vegas algorithm, this incurs an additional error along the x-axis.

snapshots $G_i^{(jm)}$ of the intermediate instances $j \in [k/m]$ of graph G_i. For *EM-CM/ES*, we start from the same seed graph, apply the algorithm and then carry out k swaps as described above.

For each snapshot, we compute several metrics, such as the average local clustering coefficient (ACC), the number of triangles, and degree assortativity.[13] We then investigate how the distribution of these measures evolves within the ensemble as we carry out an increasing number of swaps. We omit results for ACC since they are less sensitive compared to the other measures (see section 3.10.8).

As illustrated in Figure 3.11 and Appendix 3.13, all proxy measures converge within $5m$ swaps with a very small variance. No statistically significant change can be observed compared to a Markov chain with $30m$ operations (which was only computed for a subset of each ensemble due to its computational cost). *EM-HH* generates biased instances with special properties, such as a high number of triangles and correlated node degrees, while the features of *EM-CM/ES*'s output nearly match the converged ensemble. This suggests that the number of swaps to obtain a sufficiently uniform sample can be reduced for *EM-CM/ES*.

Due to computational costs, the study was carried out on multiple machines executing several tasks in parallel. Hence, absolute running times are not meaningful, and we rather measure the computational costs in units of time required to carry out $1m$ swaps by the same process. This accounts for the offset of *EM-CM/ES*'s first data point.

The number of rounds required to obtain a simple graph depends on the degree distribution. For **const** with $n = 1 \times 10^5$ and $\mu = 1$, a fraction of 5.1 % of the edges

[13]In preliminary experiments, we also included spectral properties (such as extremal eigenvalues of the adjacency/laplacian matrix) and the closeness centrality of fixed nodes. As these measurement are more expensive to compute and yield qualitatively similar results, we decided not to include them in the larger trials.

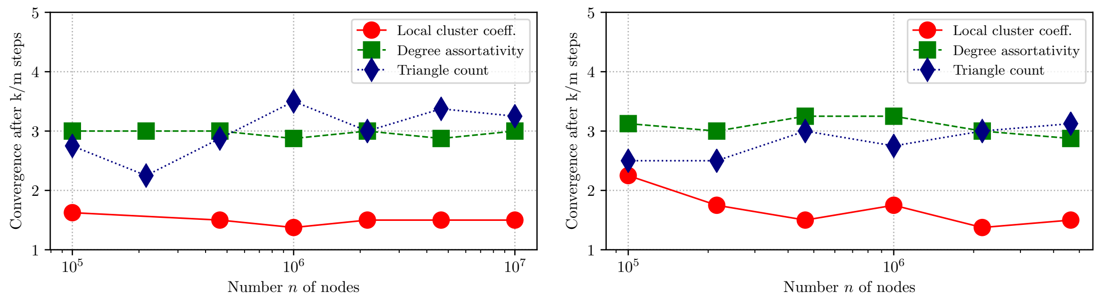


Figure 3.12: Number of swaps per edge after which ensembles of graphs with the following parameters converge: **const**, $1 \times 10^5 \le n \le 1 \times 10^7$ and $\mu = 0.4$ (left) and $\mu = 0.6$ (right). Due to computational costs, the ensemble size is reduced from $S > 100$ to $S > 10$ for large graphs.

produced by the Configuration Model are illegal. *EM-ES* requires 18 ± 2 rewiring runs in case a single swap is used per round to rewire an illegal edge. In the default mode of operation, 5.0 ± 0.0 rounds suffice as the number of rewiring swaps per illegal edge is doubled in each round. For larger graphs with $n = 1 \times 10^7$, only 0.07 % of edges are illegal and need 2.25 ± 0.40 rewiring runs.

3.10.8 Convergence of *EM-ES*

In a similar spirit to the previous section, we indirectly investigate the Markov chain's mixing time as a function of the number of nodes n. To do so, we generate ensembles as before with $1 \times 10^5 \le n \le 1 \times 10^7$ and compute the same graph metrics. For each group and measure, we then search for the first snapshot p in which the measure's mean is within an interval of half the standard deviation of the final values and subsequently remains there for at least three phases. We then interpret p as a proxy for the mixing time. As depicted in Figure 3.12, no measure shows a systematic increase over the two orders of magnitude considered. It hence seems plausible not to increase the number of swaps performed by *EM-LFR* compared to the original implementation.

3.10.9 Performance of *EM-LFR*

Figure 3.10 reports the runtime of the original *LFR* implementation and *EM-LFR* as a function of the number of nodes n and $\nu = 1$. *EM-LFR* is faster for graphs with $n \ge 2.5 \times 10^4$ nodes which feature approximately 5×10^5 edges and are well in the IM domain. Further, the implementation is capable of producing graphs with more than 1×10^{10} edges in 17 h.[14] Using the same time budget, the original implementation generates graphs more than two orders of magnitude smaller.

[14]Roughly 1.5 h are spent in the end-game of the Global Rewiring (at that point less than one edge out of 10^6 is invalid). In this situation, an algorithm using random I/Os may yield a speed-up. Alternatively, we could simply discard the insignificant fraction of remaining invalid edges.

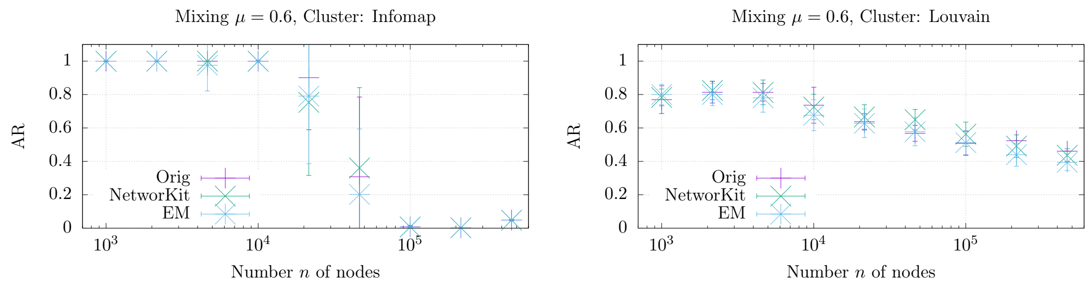


Figure 3.13: Adjusted rand measure of Infomap/Louvain and ground truth at $\mu = 0.6$ with disjoint clusters, $s_{\text{min}} = 10$, $s_{\text{max}} = n/20$.

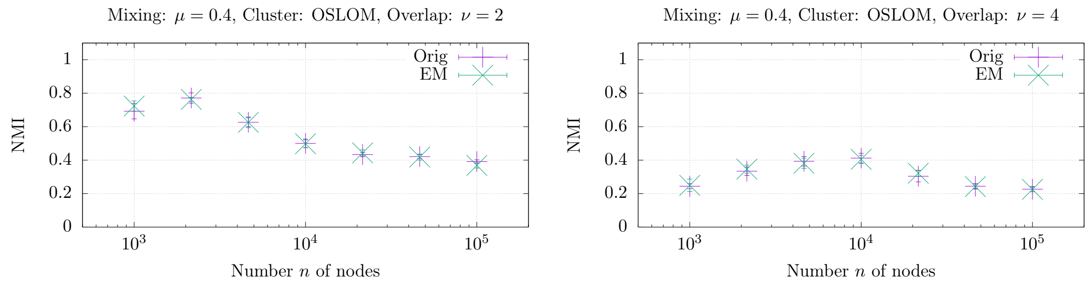


Figure 3.14: NMI of OSLOM and ground truth at $\mu = 0.4$ with 2/4 overlapping clusters per node.

3.10.10 Qualitative Comparison of *EM-LFR*

When designing *EM-LFR*, we closely followed the *LFR* benchmark such that we can expect it to produce graphs following the same distribution as the original *LFR* generator. To confirm this experimentally, we generated graphs with identical parameters using the original *LFR* implementation and *EM-LFR*. For disjoint clusters we also compare it with the implementation of NetworKit [SSM16].

For disjoint clusters, we evaluate the results of the Infomap [RAB09] and the Louvain [Blo+08] algorithm, see Section 1.4.1 for an introduction to both. We chose them as the Louvain algorithm as well as Infomap were found to achieve high quality results on LFR benchmark graphs while being fast [LF09b]. In particular the Louvain method is also among the most frequently used community detection algorithms [FH16; Emm+16].

For overlapping clusters, we evaluate the results of OSLOM [Lan+11], as OSLOM is one of the best-performing algorithms for overlapping community detection [Buz+14; FH16], see Section 1.4.4 for an introduction of OSLOM.

We compare the clusterings of the algorithms to the ground truth clusterings using the adjusted rand measure [HA85] for disjoint clusters (see also Section 2.3.2) and NMI [VR12] for both disjoint and overlapping clusters (see also Section 2.3.1).

Further, we examine the average local clustering coefficient. It measures the fraction of closed triangles and thus shows the presence of locally denser areas as expected in

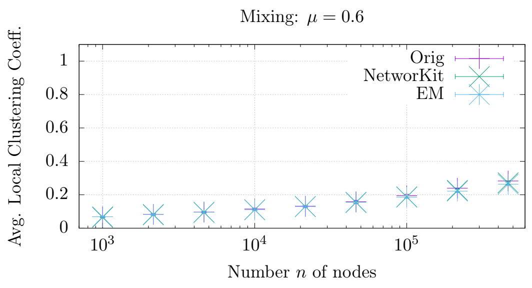


Figure 3.15: Average local clustering coefficient at $\mu = 0.6$ with disjoint clusters.

communities [Kai08], see also Section 1.3.1. We report these measures for graphs ranging from 10^3 to 10^6 nodes and present a selection of results in Figures 3.13 and 3.14 and 3.15; all of them can be found in Section 3.12 at the end of this chapter. There are only small differences within the range of random noise between the graphs generated by *EM-LFR* and the other two implementations. Note that due to the computational costs above 10^5 edges, there is only one sample for the original implementation which explains the outliers in Figure 3.13.

Similar to the results in [Emm+16], we also observe that the performance of clustering algorithms drops significantly as the graph's size grows. For Louvain, this is partially due to the resolution limit that prevents the detection of small communities in huge graphs. Due to the different power law exponents, the average community size grows much faster than the average degree as the size of the graphs is increased. Therefore, in particular the larger clusters become sparser and thus more difficult to detect with increasing graph size. On the other hand, small clusters become easier to detect as the graph size grows because outgoing edges are distributed among more nodes and are thus easier to distinguish from intra-cluster edges. This might explain why the performance of OSLOM first improves as the graph size grows. Apart from that, currently used heuristics might also just be unsuited for large graphs with nodes of very different degrees. Results on LFR graphs with one million nodes in [Ham+18a], see also Chapter 6, show that both Louvain and Infomap are unable to detect the ground truth on LFR graphs with higher values of μ even though the ground truth has a better modularity or map equation score than the found clustering. Such behavior clearly demonstrates the necessity of *EM-LFR* for being able to study this phenomenon on even larger graphs and develop algorithms that are able to handle such instances.

The quality of the community assignments used by *LFR* and *EM-LFR* is assessed in terms of the modularity $\mathcal{Q}_G(C)$ scores [NG04] (see also Section 1.3.3) achieved by the generated graph G and ground truth C. In general $\mathcal{Q}_G(C)$ takes values in $[-1, 1]$, but for large n and bounded community sizes, the modularity of a LFR graph approaches $\mathcal{Q} \to 1-\mu$ as the coverage corresponds to $1-\mu$ while the expected coverage approaches 0. For each configuration $n \in \{10^3, \ldots, 10^6\}$ and $\mu \in \{0.2, 0.4, 0.6\}$, we generate $S \geq 10$ networks for each generator and compute their mean modularity score. In all cases, the relative differences between the two generators is below 10^{-2} and for small μ typically another order of magnitude smaller.

3.11 Outlook and Conclusion

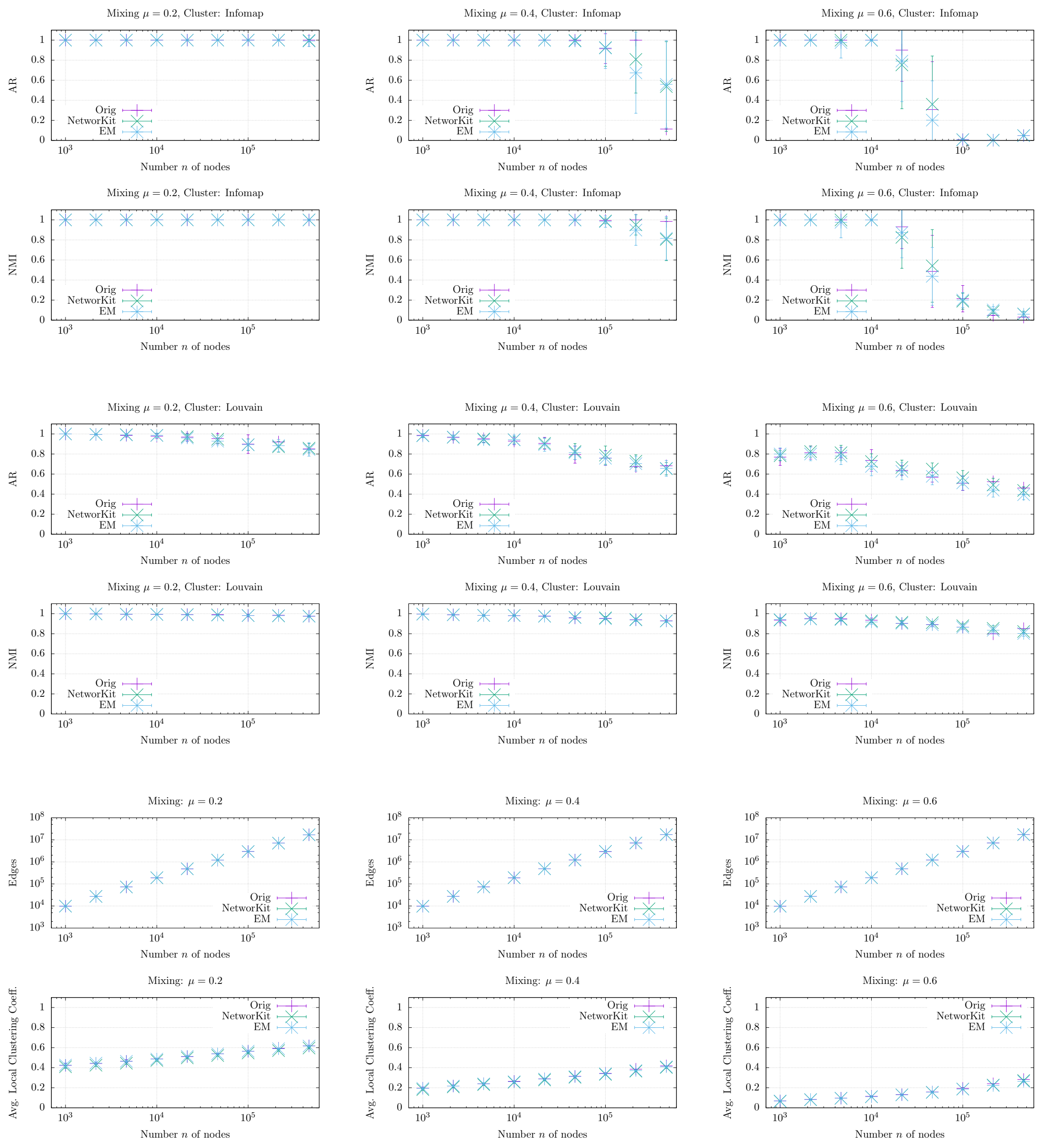
We propose the first I/O-efficient graph generator for the *LFR* benchmark and the FDSM, which is the most challenging step involved that dominates the running time: *EM-HH* materializes a graph based on a prescribed degree distribution without I/O for virtually all realistic parameters. Including the generation of a powerlaw degree sequence and the writing of the output to disk, our implementation generates 1.8×10^8 edges per second for graphs exceeding main memory. *EM-ES* randomizes graphs with m edges based on k edge switches using $\mathcal{O}(k/m \cdot \text{sort}(m))$ I/Os for $k = \Omega(m)$.

We demonstrate that *EM-ES* is faster than the internal memory implementation [VL16] even for large instances still fitting in main memory and scales well beyond the limited main memory. Compared to the distributed approach by [Bhu+14] on a cluster with 128 CPUs, *EM-ES* exhibits a slow-down of only 8.3 on one CPU and hence poses a viable and cost-efficient alternative. Our *EM-LFR* implementation is orders of magnitude faster than the original *LFR* implementation for large instances and scales well to graphs exceeding main memory while the generated graphs are equivalent. Graphs with more than 1×10^{10} edges can be generated in 17 h. We further give evidence that commonly accepted parameters to derive the length of the edge switching Markov chain remain valid for graph sizes approaching the external memory domain and that *EM-CM/ES* can be used to accelerate the process.

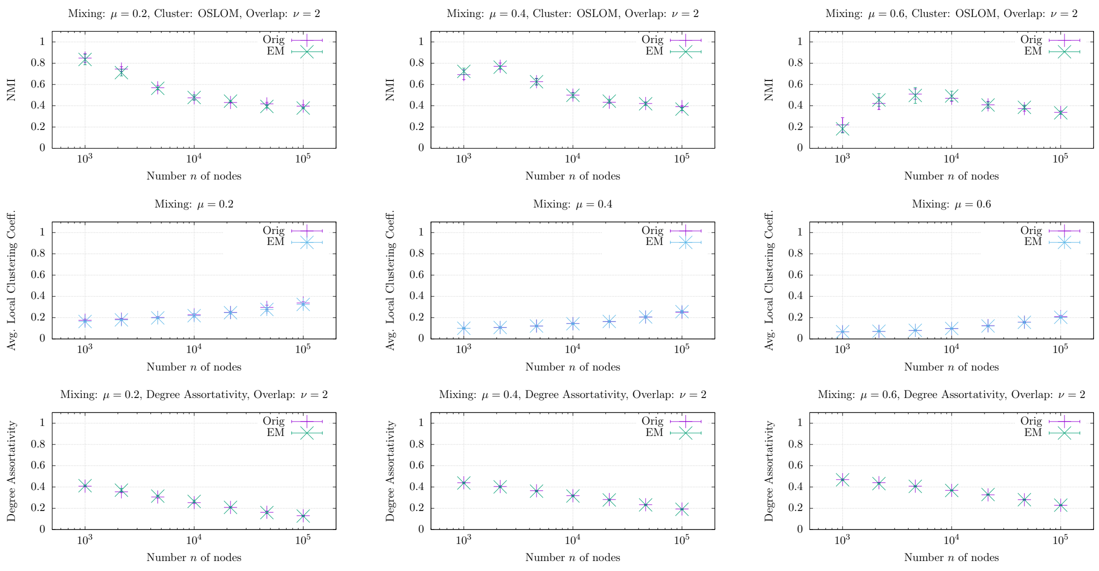
Curveball trades [CBS18] are an alternative to edge switching for randomizing the edges of a graph while maintaining the degree sequence. In each trade, the non-common neighbors of two randomly selected nodes are shuffled while keeping the degrees of the involved nodes. In a follow-up work to this research, we developed a parallel, external memory implementation of Curveball trades [Car+18]. As the main authors of this paper are Corrie Jacobien Carstens, Manuel Penschuck and Hung Tran, we only give a short summary here. As we show, power law degree distributions are challenging for Curveball as the randomization of high-degree nodes is limited when they are paired with low-degree nodes. We introduce global trades for undirected graph where every node participates in exactly one trade. Our experiments indicate that two global trades yield a roughly similar randomization as m edge-swaps. Over *EM-ES*, we achieve speedups of 5 to 14 depending on the parameter choice. To compensate for the slower convergence, the actual speedups might be half of that. Still, this should further speed up the generation of LFR benchmark graphs using external memory.

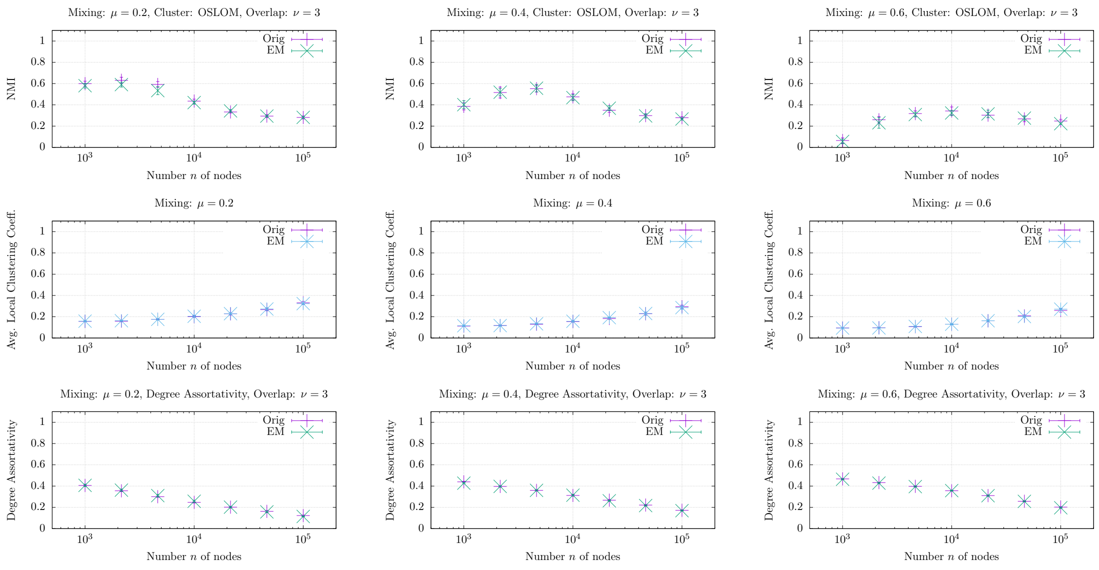
EM-LFR provides the basis for the development and evaluation of clustering algorithms for graphs that exceed main memory such as our distributed algorithms for optimizing modularity and map equation [Ham+18a] that we introduce in the next part of this thesis. There, we show that the behavior of algorithms on large graphs is not necessarily the same as on small graphs even when cluster sizes do not change, which demonstrates the necessity of such evaluations. Comparison measures such as NMI or the adjusted rand index typically do not consider the graph structure, therefore they can usually still be computed in internal memory even for graphs that exceed main memory. However, for graphs where even the number of nodes exceeds the size of the internal memory, there is the need to develop memory-efficient algorithms also for comparing clusterings.

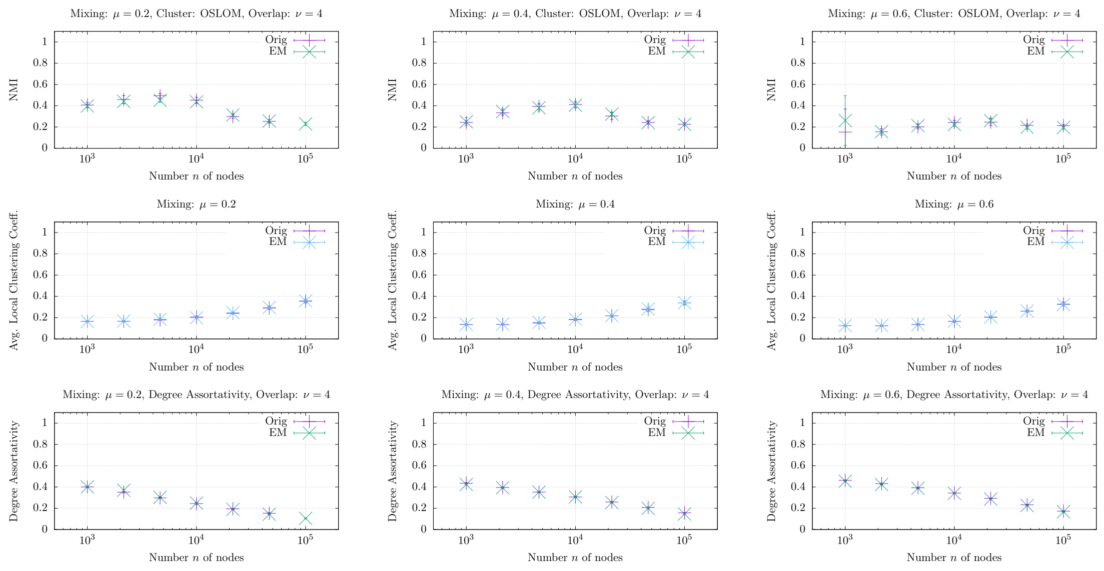
3.12 Comparing *LFR* Implementations



Comparison of the original *LFR* implementation, the NetworKit implementation and our EM solution for values of $10^3 \leq n \leq 10^6$, $\mu \in \{0.2, 0.4, 0.6\}$, $\gamma = 2$, $\beta = 1$ $d_{\min} = 10$, $d_{\max} = n/20$, $s_{\min} = 10$, $s_{\max} = n/20$. Clustering is performed using Infomap and Louvain and compared to the ground-truth emitted by the generator using AdjustedRandMeasure (AR) and Normalized Mutual Information (NMI); $S \geq 8$. Due to the computational costs, graphs with $n \geq 10^5$ have a reduced multiplicity. In case of the original implementation it may be based on a single run which accounts for the few outliers.

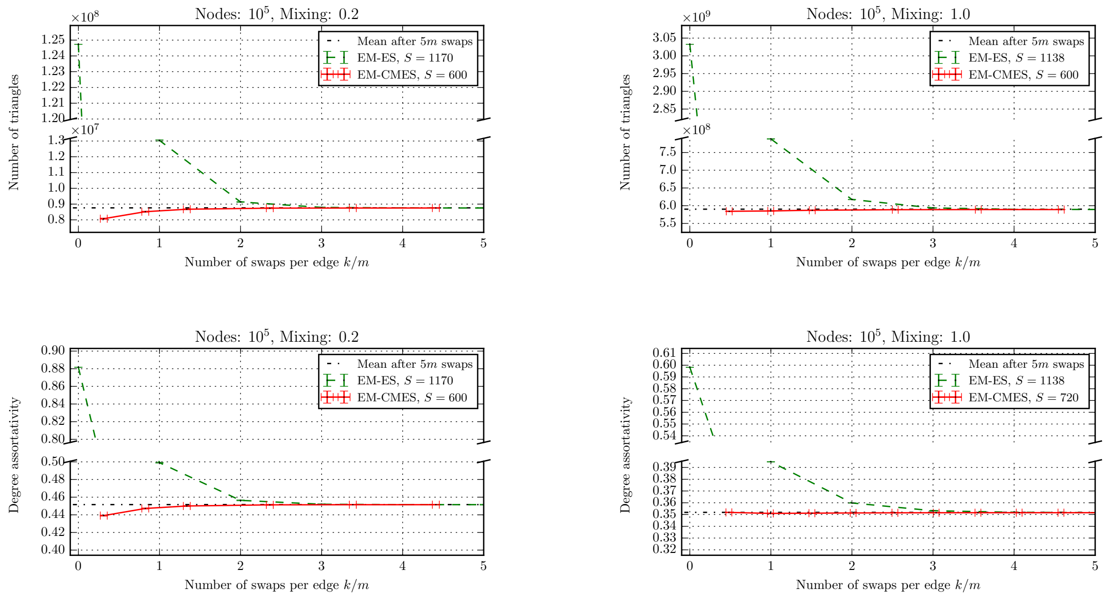

Mixing: $\mu = 0.2$, Cluster: OSLOM, Overlap: $\nu = 2$
Mixing: $\mu = 0.4$, Cluster: OSLOM, Overlap: $\nu = 2$
Mixing: $\mu = 0.6$, Cluster: OSLOM, Overlap: $\nu = 2$
NMI
Mixing: $\mu = 0.2$
Mixing: $\mu = 0.4$
Mixing: $\mu = 0.6$
Avg. Local Clustering Coeff.
Mixing: $\mu = 0.2$, Degree Assortativity, Overlap: $\nu = 2$
Mixing: $\mu = 0.4$, Degree Assortativity, Overlap: $\nu = 2$
Mixing: $\mu = 0.6$, Degree Assortativity, Overlap: $\nu = 2$
Degree Assortativity
Orig
EM
Number n of nodes
10^3
10^4
10^5

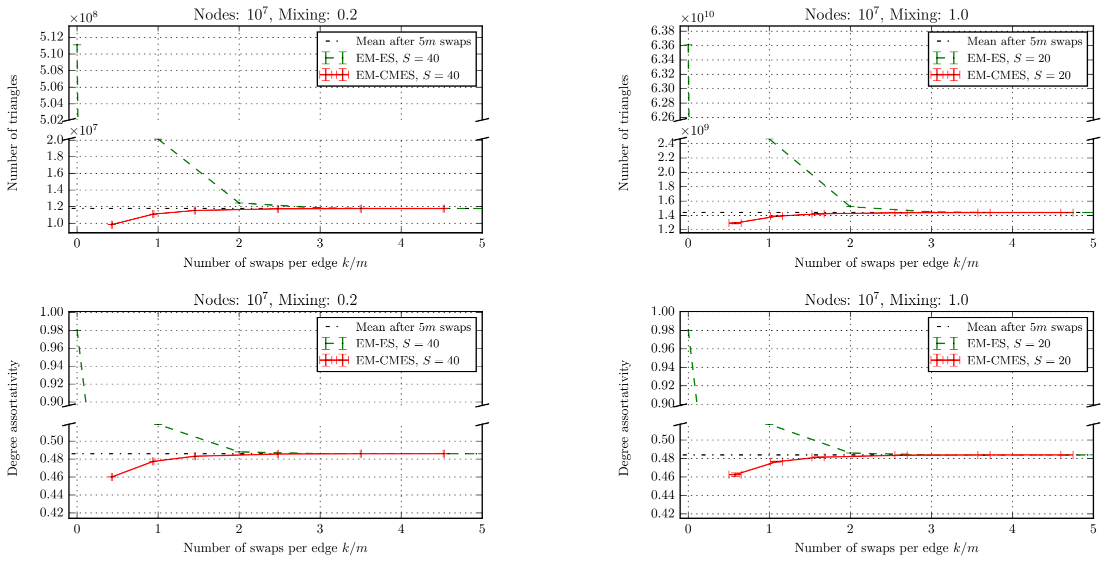

Mixing: $\mu = 0.2$, Cluster: OSLOM, Overlap: $\nu = 3$
Mixing: $\mu = 0.4$, Cluster: OSLOM, Overlap: $\nu = 3$
Mixing: $\mu = 0.6$, Cluster: OSLOM, Overlap: $\nu = 3$
NMI
Mixing: $\mu = 0.2$
Mixing: $\mu = 0.4$
Mixing: $\mu = 0.6$
Avg. Local Clustering Coeff.
Mixing: $\mu = 0.2$, Degree Assortativity, Overlap: $\nu = 3$
Mixing: $\mu = 0.4$, Degree Assortativity, Overlap: $\nu = 3$
Mixing: $\mu = 0.6$, Degree Assortativity, Overlap: $\nu = 3$
Degree Assortativity
Orig
EM
Number n of nodes
10^3
10^4
10^5



Comparison of the original *LFR* implementation and our EM solution for values values of $10^3 \leq n \leq 10^6$, $\mu \in \{0.2, 0.4, 0.6\}$, $\nu \in \{2, 3, 4\}$, $O = n$, $\gamma = 2$, $\beta = 1$ $d_{\min} = 10$, $d_{\max} = n/20$, $s_{\min} = 10\nu$, $s_{\max} = \nu \cdot n/20$. Clustering is performed using OSLOM and compared to the ground-truth emitted by the generator using a generalized Normalized Mutual Information (NMI); $S \geq 5$.

3.13 Comparing *EM-ES* and *EM-CM/ES*





Triangle count and degree assortativity of a graph ensemble obtained by applying random swaps/the Configuration Model to a common seed graph. Refer to section 3.10.7 for experimental details.

4 Benchmark Generator for Dynamic Overlapping Communities in Networks

This chapter is based on joint work with Neha Sengupta and Dorothea Wagner [SHW17]. While the model of the graph and the dynamic events have only been slightly changed compared to the publication, there have been significant changes to the algorithms used for realizing the graph. For this, large parts of the description of the algorithm as well as the experimental results have been rewritten and are based on a new implementation and therefore also new experiments.

We describe a dynamic graph generator with overlapping communities that is capable of simulating community scale events while at the same time maintaining crucial graph properties. Such a benchmark generator is useful to measure and compare the responsiveness and efficiency of dynamic community detection algorithms. Since the generator allows the user to tune multiple parameters, it can also be used to test the robustness of a community detection algorithm across a spectrum of inputs. In an experimental evaluation, we demonstrate the generator's performance and show that graph properties are indeed maintained over time. Further, we show that standard community detection algorithms are able to find the generated community structure.

To the best of our knowledge, this is the first time that all of the above have been combined into one benchmark generator, and this work constitutes an important building block for the development of efficient and reliable dynamic, overlapping community detection algorithms.

4.1 Introduction

A large portion of the existing literature on community detection overlooks at least one of two key aspects of a multitude of real world graphs, (a) their dynamic nature, i.e. edges and nodes keep getting added and deleted and (b) the often observed highly overlapping and complex structure of such networks [GDC10; YL14].

Recently proposed approaches have identified the above problems and provide solutions for detecting overlapping communities in temporal graphs [Dua+12; Ngu+11; CKU13; AP14]. However, it is difficult to empirically evaluate and compare these methods due to the lack of a realistic and fast benchmark network data generator or real-world data sets with reliably labeled, dynamic ground truth communities. As also discussed in Chapter 2, real-world data sets might provide important insights, but most of the time such data is either unavailable e.g. for privacy reasons, or does not contain reliable ground truth data to compare the found communities against. Benchmark graph generators allow to evaluate the behavior of community detection algorithms on graphs with different, predefined properties and thus test the robustness of the algorithm against a well-defined set of ground truth communities.

The requirements of a benchmark graph generator in such a scenario are diverse. Existing generators for static benchmark graphs such as LFR [LFR08] and CKB [Chy+14] replicate properties of real-world networks like that node degrees and community sizes follow power law distributions. CKB additionally ensures that the number of communities a node belongs to follows a power law distribution and has a positive correlation with the node degrees. Our goal is to extend the CKB model with a dynamic component that simulates changes in the community structure while *maintaining* these graph properties.

It is commonly agreed on [PBV07; BKT07; APU09; GDC10], that the evolution of communities can be characterized by the fundamental events birth, death, merge, split, expansion and contraction. The challenge is that such community scale events also affect the properties of the graph. For example when a new community appears, many edges need to be added to the nodes of the community which might distort the degree distribution. An analogous problem comes up when communities cease to exist, two communities merge into a single community or one community splits into two individual communities [GDC10]. Moreover, since nodes leave or join communities, it also has the potential to affect the community size distribution. Further, we want to allow fine-grained control over the rapidity with which events take place in the generated graph as most communities in a real-world setting evolve gradually over time [Bac+06]. A community detection algorithm must thus be able to follow smaller changes in order to detect large-scale changes in the community structure. This can be used to evaluate the sensitivity of different community detection algorithms. It has been shown that all of these events can also be observed in real-world networks [GDC10]. Therefore, any dynamic community detection algorithm that is supposed to work on non-trivial real-world graphs needs to be able to detect at least these basic events.

4.1.1 Our Contribution

In this work we extend the CKB benchmark graph generator for overlapping community structures to generate communities that are evolving over time. Our generator simulates community events like birth, merge, split, death, expansion, and contraction which result in node and edge insertions and deletions that gradually change the graph. A set of parameters allows controlling various properties of the graph as well as the speed of the dynamic process. In our model, at every time step, a configurable number of community events is triggered that is proportional to the number of communities. However, our generator can also be easily adapted to follow a different pattern of events.

We show using empirical analysis that the graph generator is fast and produces graphs that actually maintain the properties of the CKB model over time, i.e., the node degrees, the number of communities per node and the community sizes follow power law distributions. Further, we show that we achieve a realistic average local clustering coefficient over time. Also, we show using standard, existing community detection algorithms that our generator produces graphs whose link structure reflects its ground truth community assignment over time and that it is capable of differentiating between different community detection algorithms. A preliminary version of this work has been published as [SHW17]. Compared to this publication, we replaced the algorithms to realize the high-level community event sequence as actual graph changes. The consequence of this is that we can better preserve the properties of the benchmark graphs over time. We also changed the way community events are triggered, allowing multiple events to start at the same time. In particular for large graphs, this allows for much faster changes to the overall community structure.

Section 4.2 describes the prior work. Section 4.3 describes the algorithm in detail and Section 4.4 presents the experimental evaluation of the generation.

4.2 Related Work

For evolving networks, there is no widely used model available and those available are limited to non-overlapping communities. It is widely agreed on [PBV07; BKT07; APU09; GDC10] though, that the evolution of dynamic communities can be characterized by the following fundamental events: *birth, death, merging, splitting, expansion,* and *contraction.* In [GDC10], they describe an algorithm for tracking the evolution of a given community in a dynamic graph. For the empirical evaluation, they generate synthetic LFR graphs [LF09a] along with embedded community events of each of the above types that are applied in rapid changes of the graph. In [TB11], a different model is used for the evaluation of a dynamic community detection algorithm where nodes are initially randomly assigned to a set of k communities and some nodes change their membership in each time step. Based on the planted partition model, [Gör+12] describes an algorithm for generating a dynamic graph with non-overlapping clusters that also features the above-mentioned event types, they are slowly applied using random changes over several time steps. In a more recent work, [Gra+15] introduces another benchmark model again based on the planted partition model but only considering grow-shrink and merge-split operations. They propose three benchmarks that feature either one or both of the operations.

4.3 Algorithm

Our dynamic graph generator generates a series of simple, unweighted graphs $G_0, \ldots, G_T$ with overlapping, evolving ground truth communities $\mathcal{C}_0, \ldots, \mathcal{C}_T$ over $T+1$ discrete time steps. Each graph $G_t = (V_t, E_t)$ consists of $n_t := |V_t|$ nodes and $m_t := |E_t|$ edges. The first graph G_0 with reference communities $\mathcal{C}_0$ is directly generated according to the model of the CKB generator [Chy+14] with parameters as summarized in the first part of Table 4.1. In each of the T discrete time steps we maintain its properties:

The number of overlapping communities a node is part of follows a power law distribution $\textsc{Pld}([x_1, x_2), \beta_1)$. Community sizes also follow a power law distribution $\textsc{Pld}([n_{\min}, n_{\max}), \beta_2)$. Every community is modeled as a $G(n,p)$ graph, i.e., a graph where every edge has the same probability of existence [Gil59]. The edge probability is

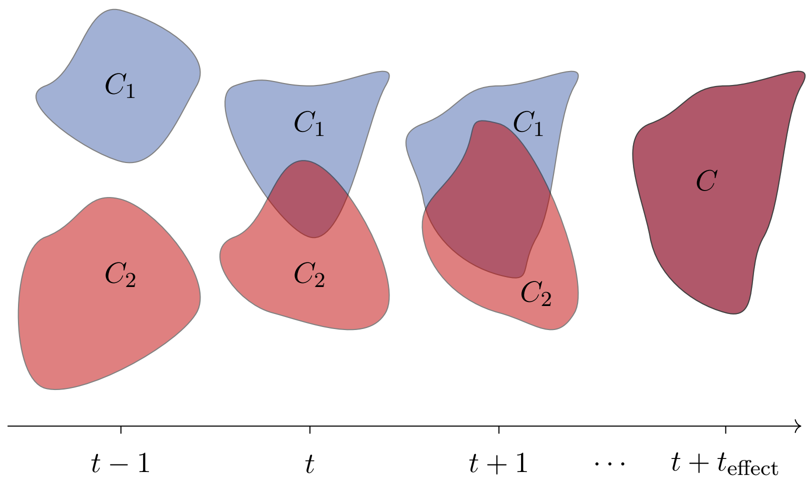


Figure 4.1: Merge of communities C_1, C_2 into C by growing their overlap.

$\alpha/|C|^{\gamma}$ and thus decreasing with increasing community size. To add noise to the graph, a so-called *epsilon community* containing all nodes with edge probability ϵ is added.

In the first part of Table 4.1, we list these parameters along with their recommended values. Except for the value of n_{min}, the recommendations are the same as in [Chy+14]. As community sizes follow a power law distribution, the value of n_{min} is the size of many communities. In [Chy+14], $n_{\text{min}} = 2$ is proposed. This would lead to a large number of communities of two or three nodes that should be detected by a community detection algorithm. Communities of size two are indistinguishable from other edges but also three nodes can hardly be called a community in most contexts. In our paper [SHW17], we propose $n_{\text{min}} = \min(n_0/100, 20)$, which leads to less than $n_0/10$ communities with the recommended parameters for larger graphs. As with these parameters a node can be part of up to $n_0/10$ communities, this made it impossible to realize the node-community assignment. While our inexact assignment proposed in [SHW17] simply assigns fewer communities to these nodes, our new assignment algorithm described in Section 4.3.4 tries harder to fulfill these requirements in every time step and thus caused slow-downs in practice. Therefore, we instead propose $n_{\text{min}} = 6$. This is a compromise between not too small communities and a large enough number of communities. In expectation, this leads to more than $n_0/10$ communities and we found that in practice the node-community assignment is exactly realizable in most cases.

Initially, in time step 0, such a graph with n_0 nodes is generated (see Section 4.3.1). In each time step $t \in [1, \ldots, T]$, *community events* may be triggered. Each community event is spread over t_{effect} time steps. In each time step, a community may only be part of a single community event.

We consider four types of community events – birth, death, merge, and split. Other events considered in the literature such as community expansion and contraction are combined with them as described below. Nevertheless, it is straightforward to add these as separate community events in our model. In a *community birth* event, a new community of

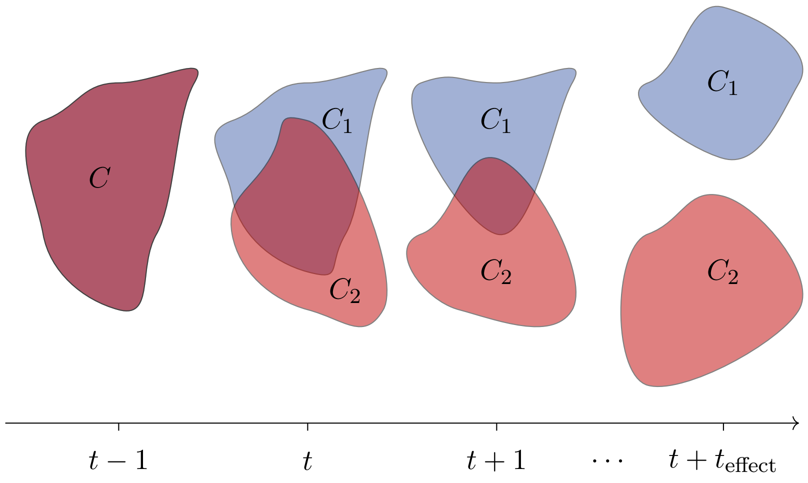


Figure 4.2: Split of a community C into C_1, C_2 by duplicating C and then reducing their overlap.

size $n_{\min}$ is created, that then grows to a size drawn from the community size distribution over t_{effect} time steps. Similarly, in a *community death* event, the community first shrinks to size $n_{\min}$ over t_{effect} time steps, before it disappears. In a *community merge* event, two randomly chosen communities C_1, C_2 grow their overlap into a new community C over t_{effect} time steps by adding nodes in $C_1 \setminus C_2$ to C_2 and vice versa. Figure 4.1 illustrates this. To avoid changing the community size distribution, we draw a new size for C from the community size distribution and either gradually remove non-overlapping nodes to shrink the final size or add additional new nodes after the overlap reaches the union of both communities. In a *community split* event, a randomly chosen community C is split into two communities C_1, C_2. We start C_1 and C_2 as exact copies of C and then gradually decrease their overlap, such that in the end both keep disjoint fractions of the nodes of C. Again, we draw new sizes for C_1 and C_2. To achieve the desired size, we either remove nodes from both C_1 and C_2 to further shrink them, or we add additional nodes to them to increase their size. Figure 4.2 illustrates the process of splitting a community.

In every time step, we collect all communities that want new members and then assign nodes to them in a *assignment step*. If communities want more members than there are nodes that want to be part of additional communities, nodes may be assigned to more communities than originally desired. If nodes want to be part of more additional communities than communities want members, we try to reduce such over-assignments by removing nodes that are part of too many communities from some of their communities.

In addition to the community events, in each time step, *node events* happen, where individual nodes are added to or removed from the graph. When a node is removed, it leaves all its communities and those communities get new members in the assignment step. New nodes simply participate in the assignment step to join communities.

The number of community and node events is proportional to the number of communities and nodes. The number of community events is drawn from a binomial distribution with

$|\mathcal{C}_t|$ trials and probability p_c. Similarly, the number of nodes events is drawn from a binomial distribution with $|n_t|$ trials and probability p_n. This ensures that with more nodes and thus also more communities, every node and community will be part of an event after a similar number of time steps. We try to balance events that destroy (death and merge) and create communities (birth and split) and nodes to avoid significant changes in the size of the graph and to avoid imbalances between the sum of the community sizes and the sum of the number of communities nodes want to be part of. Let x be the ratio between desired and actual sum of community sizes/number of nodes. Then we set the probability of events that create communities/nodes to $x/(1+x)$. The individual community events in each category (death vs. merge, birth vs. split) are equally likely.

Every time a node joins or leaves a community, we trigger *edge events*. Each community has an edge probability according to its size. When a node u joins a community C, edges from u to other nodes in C are created according to this probability. When a node u leaves a community C, edges incident to u in C are removed again. The probability is adjusted when the desired size of a community changes due to a community event. To adjust the actual edge density, we draw a desired number of edges from the binomial distribution of the number of edges in the $G(n,p)$ model of the community and remove or add edges uniformly at random accordingly. An edge $\{u,v\}$ may exist in several communities. It exists in G_t, if it exists in at least one community in $\mathcal{C}_t$ or the global epsilon community.

To smoothen community changes, edges that are created from a node u that joins a community C at time step t are already created at time step $t-x$, where x is drawn from a geometric distribution with probability λ. Similarly, edges of a node u that leaves C at time step t still exist until time step $t+x$, where x is drawn from a geometric distribution with probability λ. We recommend setting λ to a relatively high value like 0.8 to ensure that most edges are created or removed at time t where u joins or leaves C as otherwise community detection algorithms might detect u as part of C even though it is not yet or no longer part of the ground truth community C.

To create further noise and avoid that all edge changes are due to changes in the community structure, random *edge perturbations* happen in every time step with probability p_e. More precisely, at every time step in every community C, the number of edge perturbations is drawn from a binomial distribution with $|C|$ trials and probability p_e. Instead of simply removing and inserting that number of edges, we additionally draw a new desired number of edges from the binomial distribution of the number of edges and bias edge perturbations towards reaching that number. Thus, as an additional challenge for community detection algorithms, the actual density of a community might change over time.

The parameters for the dynamic events with their recommended parameters are summarized in the second part of Table 4.1.

The output of our algorithm is a stream of node, edge and community assignment events for each time step. From these events, a full graph G_t and ground truth communities $\mathcal{C}_t$ can be obtained for every time step $t \in [0, \dots, T]$.

Param.	Meaning	Recomm. Value
n_0	Number of nodes in G_0	-
$x_{\min}$	Minimum node membership	1
$x_{\max}$	Maximum node membership	$n_0/10$
β_1	Community Membership Exponent	2.5
$n_{\min}$	Minimum size of community	6
$n_{\max}$	Maximum size of community	$n_0/10$
β_2	Community Size Exponent	2.5
α	Intra Community Edge Prob. $= \frac{\alpha}{n^\gamma}$	2
γ	Intra Community Edge Prob. $= \frac{\alpha}{n^\gamma}$	0.5
ϵ	Inter Community Edge Probability	$2n_0^{-1}$
T	Number of time steps	-
p_c	Community Event Probability	10^{-2}
p_n	Node Event Probability	10^{-3}
p_e	Edge perturbation probability	10^{-2}
λ	Community event sharpness	0.8
t_{effect}	Time steps for community events to take effect	10

Table 4.1: Parameters used in the Graph Generator

4.3.1 Initialization

Our initialization step resembles the distributed algorithm for the CKB generator described in [Chy+14]. As we only consider a sequential setting, we simplified some of the steps.

First, node-community memberships and community sizes are drawn. As we draw node-community memberships, we also directly add the nodes to the epsilon community. Then nodes are assigned to communities in the *node-community bigraph*, see Section 4.3.4 how realize this graph. Whenever we add a node to a community, the corresponding edges are generated as described in Section 4.3.5.

4.3.2 Community Events

In the following, we describe for each of the four community events – birth, death, merge and split – how it is implemented. In this section, we only deal with node-community memberships and changes of the desired size. Edges are always implicitly generated as explained in Section 4.3.5. Further, instead of directly adding new nodes to a community, we only increase its desired size. In the community assignment phase that is described in Section 4.3.4, nodes are assigned to communities such that their desired sizes are met.

Community Birth

To spawn a new community C at time t, we first draw its desired final size n_C from $\textsc{Pld}([n_{\min}, n_{\max}), \beta_2)$. At time t, C starts with $n_{\min}$ desired nodes. Then, in each of the following time steps $t + t'$, $t' \in [0, t_{\text{effect}})$ an expansion phase follows where the desired size is increased by

$$\left\lceil \frac{n_C - |C_{t+t'-1}|}{t_{\text{effect}} - t'} \right\rceil .$$

The birth event ends when the desired size reaches n_C nodes, which happens at the latest at time step $t + t_{\text{effect}} - 1$.

Community Death

Symmetrical to the expansion phase after the birth event, the community death event is preceded by a contraction phase, where nodes gradually leave the community until only a core of the community remains. In a death event starting at time step t, in each time step $t + t'$, $t' \in [0, t_{\text{effect}} - 1)$, we sample nodes uniformly at random to leave the community. The number of nodes to remove is

$$\left\lceil \frac{|C_{t+t'-1}| - n_{\min}}{t_{\text{effect}} - t' - 1} \right\rceil .$$

If the community size is equal to $n_{\min}$, which is the case at the latest at time $t + t_{\text{effect}} - 1$, we remove all nodes and the community dies.

Community Split

A community split events starts by creating a duplicate of the input community C such that we have now two identical communities C_1, C_2. Note that this duplicate also inherits an exact copy of the edges of C, i.e., there is no change in the graph due to this duplication. Further, we draw desired final sizes n_{C_1}, n_{C_2}. Recall that over the course of the split event the nodes of C shall be divided among C_1 and C_2, i.e., their overlap is gradually removed. For this, we initially determine lists of nodes R_1, R_2 to remove from C_1 and C_2 over the course of the t_{effect} time steps. If $n_{C_1} + n_{C_2} < |C|$, we first randomly select $|C| - n_{C_1} - n_{C_2}$ nodes, add them to R_1 and R_2 and then assign the remaining nodes to C_1 and C_2 by adding all other nodes to R_2 and R_1. Otherwise, we assign fractions of nodes proportional to $n_{C_1}/(n_{C_1} + n_{C_2})$ and $n_{C_2}/(n_{C_1} + n_{C_2})$ of C to R_2 and R_1, respectively. We round randomly by drawing a value $r \in [0, 1)$ and rounding up if the fractional part is larger than r.

Similar to community birth and death events, in each time step, we now both gradually remove nodes from the list of nodes to be removed and simultaneously increase the desired size to finally reach n_{C_1} and n_{C_2} nodes, respectively, after t_{effect} time steps. In time step $t + t'$, we first remove

$$\left\lceil \frac{|R_i|}{t_{\text{effect}} - t'} \right\rceil$$

nodes of R_i from each community C_i, $i \in \{1,2\}$ and set $R_i := R_i \cap C_i$. Then, we increase the desired size of C_i to

$$\max\left\{|C_i| + \left\lceil \frac{n_{C_i} + |R_i| - |C_i|}{t_{\text{effect}} - t'} \right\rceil, n_{\min}\right\}$$

for $i \in \{1,2\}$.

Note that while C_1 and C_2 might be completely disjoint after the split event, this is not necessarily the case as nodes that were previously part of C might be re-added to C_1 or C_2 to achieve the desired size during the node-community assignment.

Community Merge

The merge event follows a process that is almost the reverse of the process of the split event. We gradually increase the overlap between two communities C_1, C_2 until they merge into a new community C. For this, the nodes of C_1 start joining C_2 while the nodes of C_2 join C_1. Recall that C has a new size n_C that might be larger or smaller than the union of C_1 and C_2. If the resulting community shall be smaller, we remove some of the nodes of C_1 and C_2 that are not yet in the overlap. Further, if the merged community has not yet the target size n_C, we gradually increase or decrease its size to n_C.

The merge event first calculates the overlap between C_1 and C_2 and creates list of nodes $A_1 := C_2 \setminus C_1$, $A_2 := C_1 \setminus C_2$ of nodes that shall be added to the respective communities. At each time step $t + t'$, $t' \in [0, t_{\text{effect}})$, we add

$$a := \max\left\{0, \left\lceil \frac{n_C - |C_1 \cap C_2|}{t_{\text{effect}} - t'} \right\rceil\right\}$$

nodes to the overlap and remove

$$r := \max\left\{0, \left\lceil \frac{|C_1 \cup C_2| - n_C}{t_{\text{effect}} - t'} \right\rceil\right\}$$

nodes from the non-overlapping part.

We first remove $\min\{r, |A_1 \cup A_2|\}$ nodes from the non-overlapping parts of C_1 and C_2 and A_1 and A_2. If $A_1 \cup A_2 = \emptyset$, the merge finished, i.e., $C_1 = C_2$ and we simply remove C_2. If we have removed less than r nodes so far, we are now after the merge and remove the remaining nodes from C_1.

If we are after the merge and $a > 0$, we simply increase the desired size of C_1 by a, but at least to $n_{\min}$. If C_2 still exists but either $|C_1| + |A_1| \leq n_{\min}$ or $a \geq |A_1| + |A_2|$, we increase the desired size of C_1 to $|C_1| + a$, but at least $n_{\min}$, add all nodes in A_1 to C_1 and perform the merge by removing C_2. Otherwise, we select a_1 and a_2 nodes from A_1 and A_2 to add to C_1 and C_2, respectively. We choose a_1 and a_2 such that both communities reach at least $n_{\min}$ nodes. If we still selected less than a nodes, we select the remaining nodes proportional to the sizes of A_1 and A_2, respectively. We then increase the desired sizes of C_1 and C_2 by a_1 and a_2 and add the corresponding nodes from A_1 and A_2 to them.

4.3.3 Node Events

In every time step, we possibly add or remove one or several nodes. Removed nodes immediately leave all communities they are part of. While in most cases they are simply replaced by another node during the node-community assignment phase, there are a few special cases if the community is part of a community event. For community birth events, nothing changes, the community continues to grow, and we simply assign more nodes during the node-community assignment phase. For community death events, we do not replace removed nodes. This means that the community shrinks faster. For community split events, we also do not immediately replace removed nodes, instead, we remove them from the corresponding set R_i and, if necessary, let the community grow more to replace them. Community merge events are similar, there we also remove the node from the corresponding set A_i and possibly let the community grow more after the merge has been completed.

4.3.4 Node-Community Assignment

After all active community events have been processed and all node events happened, we ensure that every community has as many members as desired. For this, we build the *node-community bigraph*, a bipartite graph between the nodes V_t and the communities in $\mathcal{C}_t$. An edge between a node $u \in V_t$ and a community $C \in \mathcal{C}_t$ indicates, that we freshly assign u to C in time step t. We build this assignment in several steps. The first step is to reduce over-assignments if there are any and we can possibly replace over-assigned nodes by other nodes. If necessary, we sample additional nodes or exclude nodes from the assignment to match the number of nodes wanted by communities. To assign the nodes, we first compute a greedy assignment of nodes to communities. Then, we shuffle those assignments. The node-community assignment ends by actually adding all freshly assigned nodes to their communities.

First, we calculate the sum s_V of the additional communities nodes want to be in and the sum s_C of the additional members all communities want. If $s_V > s_C$, we try to remove some nodes from communities to be able to assign more nodes to communities. For this, we keep track of over-assigned nodes, i.e., nodes that are part of more communities than they want to be part of. We iterate over these nodes in a random order. For each node u, we select communities uniformly at random and remove u from them until u is only part of its desired number of communities or $s_v = s_C$. Note that this only increases s_C but not s_V, as we are only reducing over-assignments. To not to disturb the creation or removal of overlap in community split and merge events, we exclude all communities that are currently part of a split event or a merge event before the merge happened. If s_V is still larger than s_C, we select a uniform sample of nodes that will get fewer communities than originally wanted. To avoid that a node has no community, we first select every node that has no community to get one community and only afterwards consider other nodes or getting more than one community.

In the opposite case, if $s_V < s_C$, we sample $s_C - s_V$ additional nodes that will be assigned to communities. For this, we draw uniformly at random from all nodes weighted by their desired number of communities. We do not consider how many communities a node already has or shall get, nodes may be sampled several times.

Assigning nodes to communities means generating a random bipartite graph from a prescribed degree sequence. The algorithms are similar to those for the fixed degree sequence model described in Section 3.1.1. For the greedy assignment, we use a bipartite variant of the Havel-Hakimi algorithm similar to the one described by Kleitman and Wang [KW73] that is based on the Gale-Ryser theorem [Gal57; Rys57]. Note that these algorithms originally assume that we are creating all edges from scratch. Apart from the initial assignment, this is not true. The problem of finding additional edges that satisfy certain additional degrees is the same as finding a subgraph with given degree sequence, a so-called f-factor, in the complement graph. This problem can be solved in time $O(n^3)$ in general graphs [Ans85]. For bipartite graphs, this can be simplified to solving a maximum flow problem. We found that in practice for our suggested parameters, our slightly adapted Havel-Hakimi algorithm (see below) usually still produces a valid assignment, and if not, only few assignments are missing that we fulfill by sampling additional nodes. Therefore, we did not implement the exact assignment based on f-factors that might easily dominate the running time.

We start the greedy assignment by sorting all selected nodes by the number of missing memberships and all communities that want new members by the number of missing members using bucket sort. We iterate over the selected nodes, starting with those that have the highest number of missing memberships. For each node, we iterate over all communities that want new members starting with the ones that still want most members. Whenever we find a community the node is not yet part of, we add an edge to the node-community bigraph until the node is part of as many communities as desired or we tried all communities. We store communities in a bucket priority queue that we update as nodes are assigned to communities. For the initial assignment, we could use the technique explained in Section 3.5 to avoid changing the order of elements in the priority queue, but for the later steps this is not possible anymore as we possibly cannot add a node to a community. If we cannot find enough communities for a node, we simply continue with the next node. After finishing this initial assignment for all nodes, we sample additional nodes and assign them directly to the communities that still need members. If we initially excluded nodes due to $s_V > s_C$, we first try them in a random order. If there are no (more) such nodes, we sample from all nodes as for the over-assignment. This sampling continues until all communities have enough members.

For the shuffling of the assignment we use edge switching [KTV99]. For each switch, we randomly select two node-community assignments. If we selected two different nodes and two different communities, we swap their nodes if none of the nodes is already part of the other community or shall be assigned to it. We perform ten times the number of assignments switching attempts as results on simple graphs with given degree sequence suggests that this is enough [RPS15]. Note that this does not necessarily yield a random assignment apart from the initial assignment step. Consider the two node-community assignments shown in Figure 4.3. While both are valid results for the same input, there is no switch possible and thus no way for our switching algorithm to get from one to

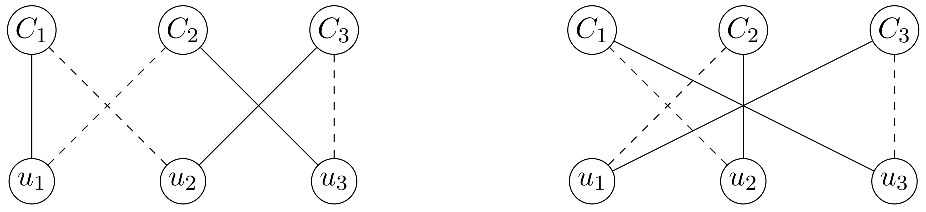


Figure 4.3: Example for two different node-community assignments with existing fixed assignments (dashed) such that there are no two assignments that can be switched to get from one to the other.

the other. As many nodes are just part of a single community and there are many small communities, we expect such problems to be rare in practice.

4.3.5 Edges

In the CKB model, and thus also in our generator, every edge belongs to one or more communities. In this section, we describe how edges within a community are generated, without caring about possible duplicates. In the following section we describe how we generate a stream of global edge events from them.

The methodology described for the distributed CKB generator [Chy+14] for generating the edges within a community involves drawing the number of edges per node to insert from a binomial distribution with success probability $p_c = \frac{\alpha}{{n_c}^{\gamma}}$ where n_c is the size of the community C, and thereafter using the configuration model. We use the Batagelj Brandes model [BB05] to more efficiently generate an Erdős-Renyi graph for each community. Every time we add a node to a community, we use this model to generate edges to existing nodes according to the current edge probability in the community. When a node is removed, its edges are removed, too.

Changing Edge Probabilities. Whenever the desired size of a community is changed, we also adjust its edge probability. To adjust the actual density of the community to the new edge probability, we first draw a new number of edges from a binomial distribution with the maximum number of edges trials and success probability equal to the new edge probability. We then add or remove random edges to reach the sampled number of edges. I.e., we simulate the $G(n, p)$ model using the $G(n, m)$ model. To remove edges, we simply perform a selection uniformly at random of the existing edges. To add edges, we sample random node pairs until we have found enough node pairs that did not exist yet. To ensure that we find a suitable node pair after a constant number of samples in expectation, we also store non-edges explicitly if after adding edges more than 75% of the node pairs exists. We stop storing them once the number of edges after removing edges decreases below 25% of the edges. This ensures that before we iterate over all node pairs, at least half of them have been added or removed. If non-edges are stored, we instead sample from them to add edges.

Edge Perturbations. At every time step, each community perturbs its edges with probability p_e. We draw the number of edges to perturb p from a binomial distribution with the current number of edges trials and probability p_e. In order to create even more noise, we do not simply replace p_e edges by new edges, but instead also sample a new number of edges as described above, just without changing the edge probability. Let x be the current number of edges and y be the new number of edges. If $y \geq x$, we add p edges and, if $p > y - x$, we remove $p - y + x$ different edges. Otherwise, we remove p edges and, if $p > x - y$, we add $p - x + y$ different edges. This is only possible if the number of edges to add is at most the number of missing edges in the community, as otherwise we cannot add enough new edges that are different from the edges to remove. If the number of edges to add is z edges larger than the number of missing edges, we add and remove z edges less. We first select a random sample of edges to remove using Fisher-Yates shuffle [FY48] on the edges of the community until we have shuffled enough edges. Then we add the desired number of edges and only afterwards remove our selected sample to ensure that we do not re-add an already removed edge. The sampling for adding edges works as described above and is thus linear in the number of added and removed edges in expectation.

Epsilon Community Every node is added to the epsilon community and its edges are created as described above. The only difference is that the edge probability is fixed and not tied to its size. Edge perturbations are also realized in the exactly same way as described above.

4.3.6 Event Stream Generation

Every time a node is added to or removed from the graph, a node joins or leaves a community or an edge is added or removed from a community, we generate an event for our global event stream. We first collect these events in buckets per time step and event type. Further, we store for every node its birth time. Note that we never revive nodes, every time a node is born we assign a fresh node id. While most events are generated in order, this is not true for edge events. As described before, when a node joins or leaves a community, depending on parameter λ, the edges that are added to or removed from the graph may already exist earlier and longer. For this, we simply add those events to an earlier or later time bucket.

As the last step, our generator creates the final event stream in a sweep over these collected events. This has several purposes. First, due to our smooth community changes, it might happen that we generated an event for adding an edge $\{u, v\}$ before u and v are born. When we encounter such an event, we move that event to the time step when both nodes exist. Also, when a node dies, we ensure that we immediately remove all incident edges. For this, during our sweep, we keep track of the incident edges of every node. Due to community overlaps also with the epsilon community, edges might be created and removed multiple times. During our sweep, we keep a counter for every edge of how many communities it is part of. For our final event stream, we only generate a birth event if the edge did not exist before in any community and a death event if it no longer part of any community. It may happen that a node is removed from and re-added to the same community within a time step, e.g., during a split event or while reducing over-assignments. For the final event stream, we only generate a community event if the membership of a node changes from before the time step to after the time step.

4.3.7 Time Complexity

Almost all operations in our generator need constant time in expectation per internally generated event. To achieve this, we use hash maps or hash sets in many places. The only exceptions are the node-community assignment step and the edge perturbations. For the node-community assignment step, running time linear in the number of all node-community assignments can be necessary due to much more nodes wanting to be part of communities than communities wanting nodes from which we need to sample or over-assignments that we try to reduce but cannot reduce due to all communities being part of community split or merge events. For edge perturbations, we draw the desired number of perturbations for each community, which might be zero and thus not generate any events. By using clever sampling techniques and keeping track of more information, it might be possible to reduce this complexity. In practice for our recommended parameters, we found that neither the node-community assignment step nor the edge perturbations are dominated by this overhead, though. The relation between internal events and the final event stream depends on many parameters, e.g., how dense communities are and their overlaps, as we remove duplicate edges. For the proposed parameters, the majority of the nodes is only part of a single community and thus overlaps are minimal. Let a be the maximum number of node-community assignments and b the number of internal events. Then the expected running time is in $O(a \cdot T + b)$.

4.4 Experiments

We present the results of several experiments conducted with our dynamic graph generator. The empirical evaluation of the generator has been done along three main categories.

1. Running Time: We measure the efficiency of the generator itself in constructing graphs with increasing graph size and event probability. The goal of these experiments is to illustrate that the generator scales well when generating large, dense, or highly dynamic graphs.

2. Graph Properties: As the dynamic graph evolves over time due to the edges and nodes changed by the generator, properties of the graph are measured at different time steps. This category of experiments is targeted at showing that the generator is capable of *maintaining* graph properties while generating community scale events.

3. Community Detection Algorithm: Finally, we use existing community detection algorithms, to detect communities on the generated datasets. We show using these experiments that the link structure of the generated dynamic graph follows the ground truth as the graph evolves.

Our primary contribution is that we add dynamic events to an existing static graph generator for overlapping communities. The goal of our experiments is, therefore, to show that generating community events on the initial graph (G_0), generated by the static generator, does not adversely affect the graph properties that G_0 exhibits. The static graph model used for generating G_0 in this work, CKB, has been shown to resemble real world graphs in terms of graph properties in [Chy+14]. We show that, in spite of the dynamic events generated on the original static graph, the same graph properties are followed and the link structure indicated by the community assignment is maintained. This implies that the dynamic generator model we propose, is able to generate a graph with evolving overlapping communities, realistic graph properties, and community events at *each* time step – forming a suitable input to a dynamic community detection algorithm.

4.4.1 Setup

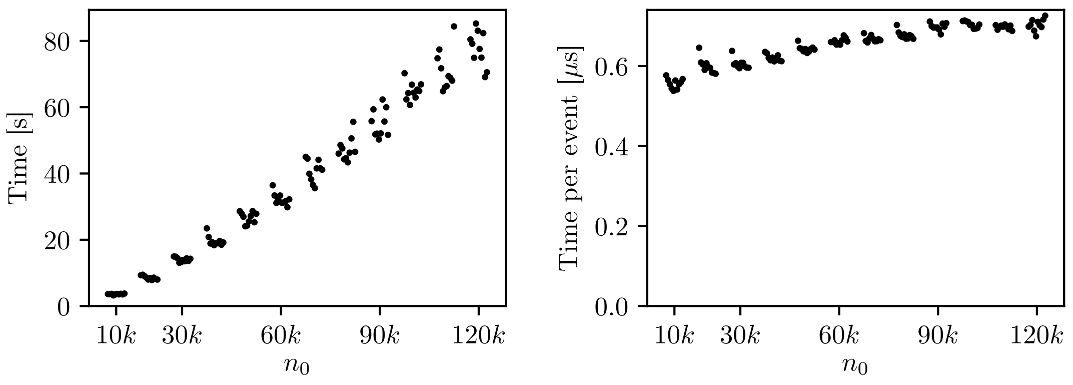
We implemented the generator in C++ in NetworKit [SSM16], the code is available at `https://github.com/michitux/networkit/tree/ckbdynamic`. In many places, we use hash maps to efficiently store nodes, edges or communities. For this, we use an implementation of robin hood hashing[1] [Cel86]. Our experiments were conducted on a system with two 8-Core Intel Xeon(tm) Skylake SP Gold 6144 processors and 196 GB 2666MHz DDR4 RAM. Our implementation is single-threaded, we thus restricted our experiments to use just a single CPU core. The parameters experimented with include n_0, the starting number of nodes in the graph, p_c, the probability of a community event occurring, α, which affects the intra cluster edge density, and ϵ, which affects the density of inter-community or global edges. Other parameters are set to their values in Table 4.1.

4.4.2 Running Time

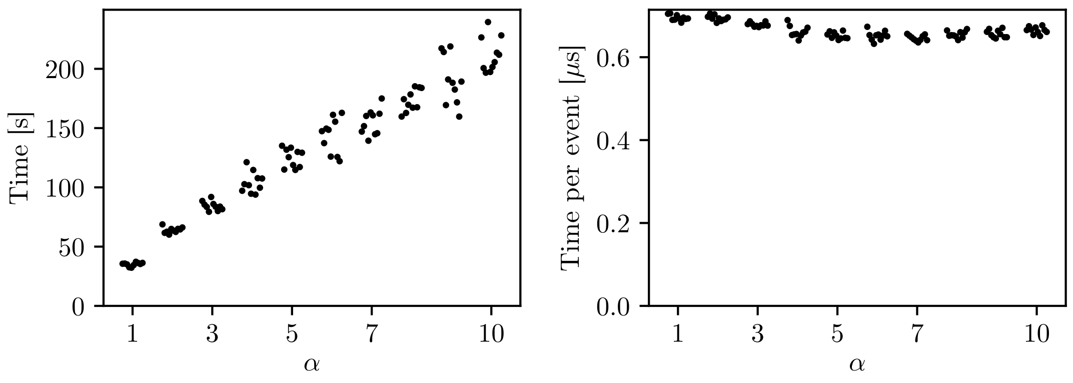
Figure 4.4 plots the time our generator needs both as absolute running time and as running time per produced event for 1000 time steps. Apart from the varied parameter, all parameters are set to the recommended values in Table 4.1. For each value, we run the generator with ten different seeds. We plot those ten runs with slightly varying x-value to separate measurements.

Figure 4.4a shows the time taken for varying number of nodes in the starting graph. As the number of nodes increases, the running time increases slightly more than linearly. This is because with n_0, also the maximum community size and the maximum number of memberships per node increases. Larger communities contain more edges and more memberships per node lead to more communities in general. Both lead to higher average degrees and the latter also leads to more community events per node, which explains the more than linear increase in running time. If we consider the running time per event, instead, the running time increases only slightly till 90k nodes and remains essentially constant for larger graphs. A possible explanation for this slightly increasing running time per event is the fact that while memory accesses are constant in theory, cache hierarchies

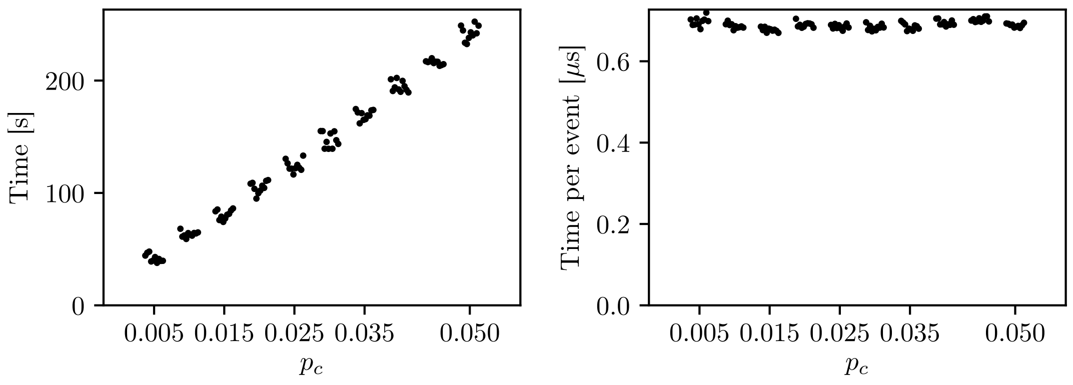
[1] `https://github.com/martinus/robin-hood-hashing`



(a) Varying starting number of nodes



(b) Varying intra cluster edge density



(c) Varying community event probability

Figure 4.4: Time taken to generate 1000 time steps, both as absolute running time and normalized by the number of events. We apply a slight jitter on the x-axis to separate measurements.

lead to increasing memory access times for larger data structures. These results show that a graph with 120k nodes and 1000 time steps can be computed in around 80 seconds.

Figure 4.4b shows the individual running times for varying values of α and 100k nodes. Here we can see that the running time is increasing slightly less than linearly and the running time per event is even decreasing with increasing density. There are several explanations for this. With increasing density, each change in the node-community assignment produces more events, and also edge perturbations increase linearly. However, with larger densities also more and more of the small communities become cliques. Our smallest communities of size 6 are cliques starting with $\alpha = 3$, communities with 100 nodes are also cliques at $\alpha = 10$. Once a community is a clique, its density cannot increase further with increasing α and also no more edge perturbations can happen. This explains part of the less than linear increase in running time. The other explanation is that there is a certain overhead like the node-community assignment or drawing the number of edge perturbations per community that is independent of the density. Tests on individual parameter choices show that neither of those dominates the running time, but still their relative weight decreases with increasing edge density.

Figure 4.4c shows the time taken for varying probabilities of community events and 100k nodes. Note that every event needs up to 10 time steps and half of the events also concern two communities. Thus, an event probability above 5% is likely to cause all communities to be part of an event. The increase in running time is again roughly linear in the event probability, as we can expect that for event probabilities above 1%, the majority of the events in the final event stream are triggered by community events. The running time per event is almost constant. The slight variations might be explained by the randomness of the generator. For small event probabilities slightly higher running times can also be explained by the parts of the generator that are independent of community events and by the fact that the node-community assignment has certain steps that are independent of the number of nodes to be assigned.

4.4.3 Graph Properties

In this section, we examine the node degrees, community sizes and node-community memberships for a dynamic graph with 10k nodes and the other parameters chosen according to the values in Table 4.1. All results in this section are from the same run of the generator, thus graphs can be compared across different plots.

In Figures 4.5, 4.6, and 4.7 we plot the degree, community size and node membership distribution at time 0, 500 and 1000. The community sizes in Figure 4.6 clearly follow power law distributions at every time step. There is some variation, which might also be explained by events that are currently in progress, but overall the distributions are very similar. The degree distribution in Figure 4.5 shows clear differences between the time steps and is also not a pure power law distribution. As node degrees are not explicitly drawn, this is also not expected. Still, also already shown for the CKB generator [Chy+14], the tail follows a power law distribution. There are some effects visible though that ask for further explanation, in particular the many nodes having a degree between 50 and 100 at $T = 0$ and $T = 1000$. The explanation for this is that in particular at $T = 1000$, there are

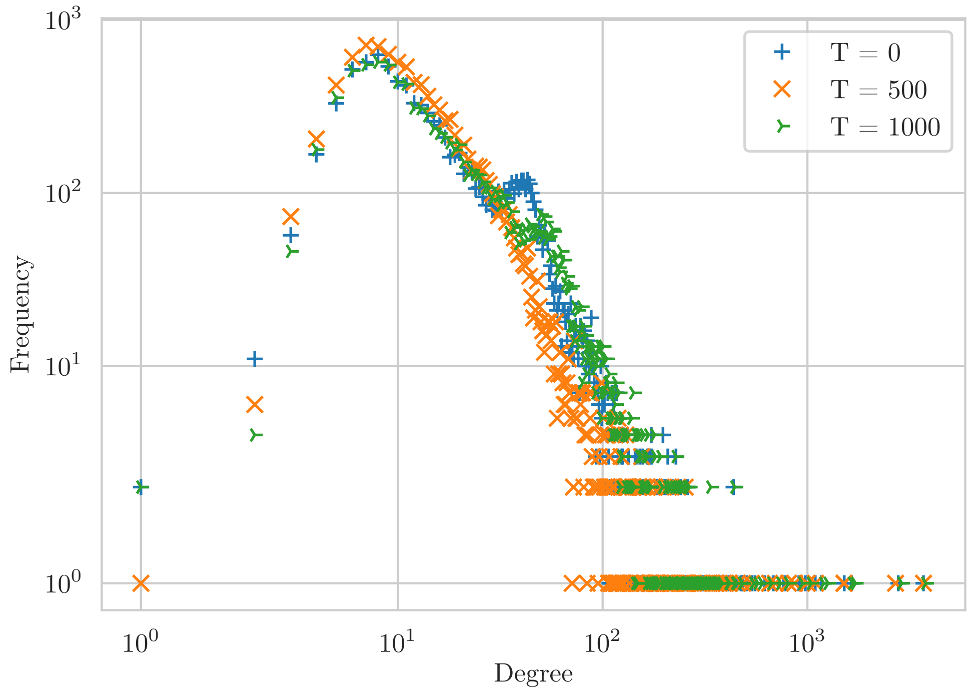


Figure 4.5: Degree Distribution

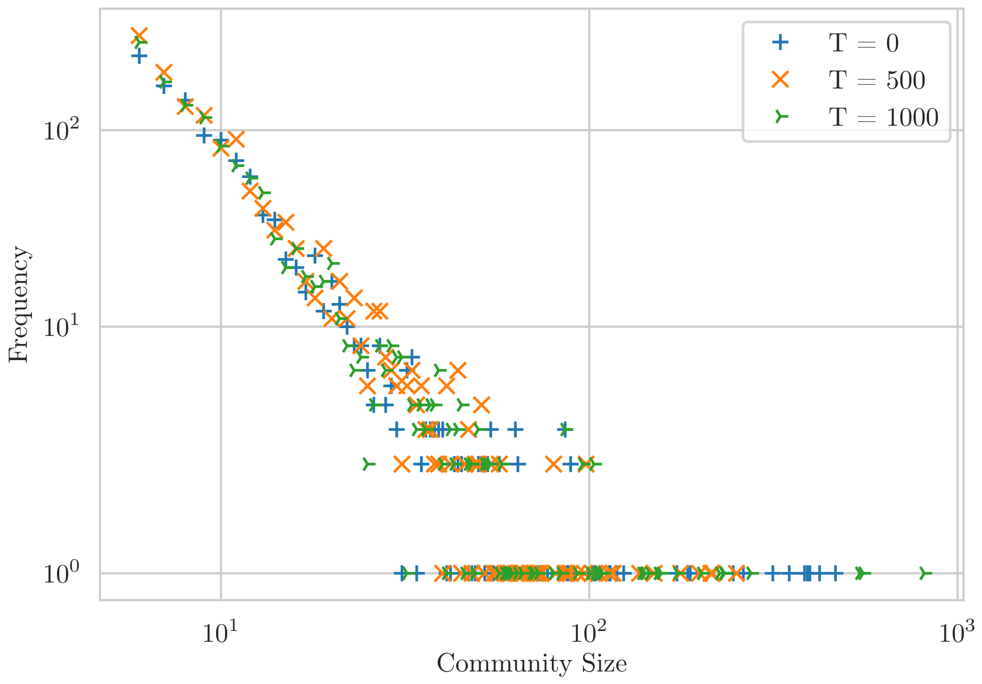


Figure 4.6: Community Size Distribution

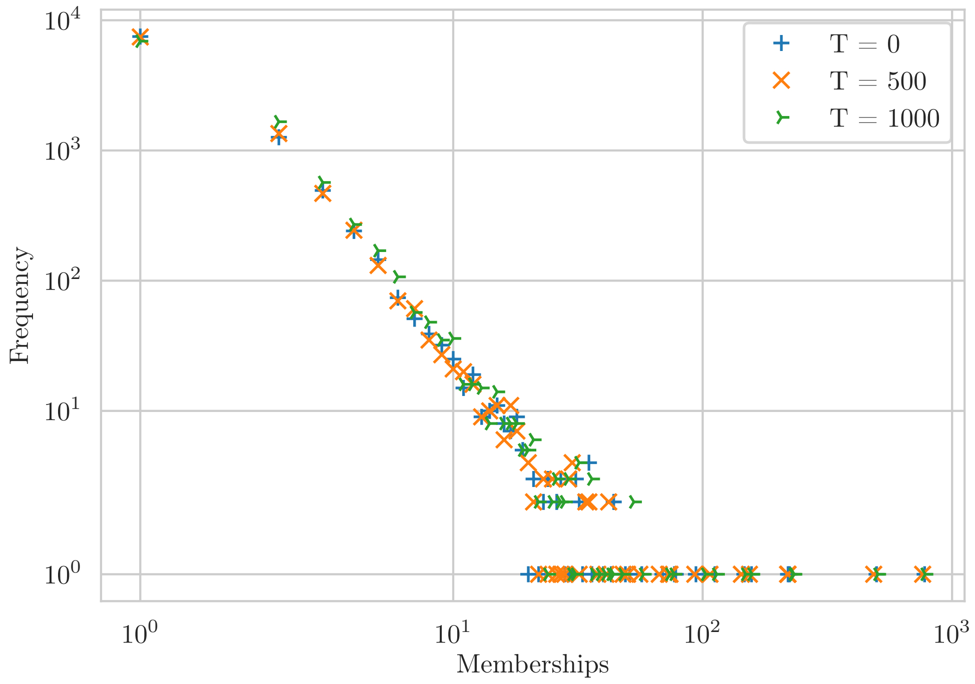


Figure 4.7: Node Membership Distribution

two huge communities of almost a thousand nodes as visible in Figure 4.6. The average degree in a community of a thousand nodes is 63, which explains the peaks. At time step 500, there are no such huge communities and thus no such peaks. The node-community membership distribution in Figure 4.7 shows almost identical power law distributions for $T = 0$ and $T = 500$. At $T = 1000$, we see a slightly shifted distribution – there are fewer nodes with a single community, but more nodes with more than one community. This is because of the over-sampling due to communities wanting more members than nodes wanting communities. As we can see, the over-sampling is quite uniform and still yields a power law distribution.

Note that a community event probability of 1% as used in these experiments means that the probability of a community to not to be part of any community event is already below 1% after 500 time steps and below 10^{-5} after 1000 time steps. Therefore, after 1000 time steps, we can safely assume that almost all communities are the result of a community event. Further, the node event probability of 0.1% means that also a large part of the nodes will have been replaced. This shows that our community and node event model are able to produce highly dynamic graphs while preserving distributions of key characteristics. In the following we also consider different properties of the graph over time.

In Figure 4.8 we show how the total number of memberships as well as the number of nodes with 0 (orphans), 1 and 2 communities varies over time. There are no orphans

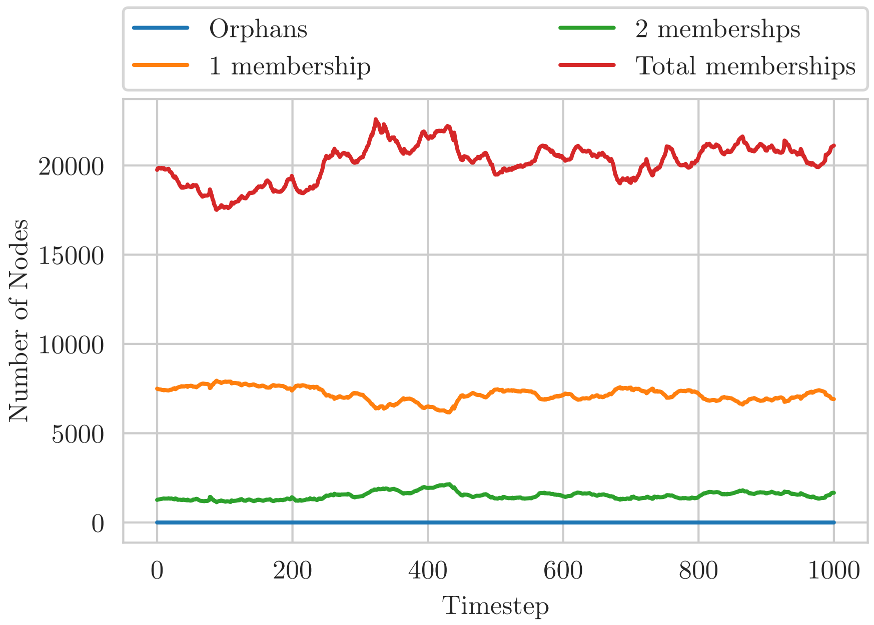


Figure 4.8: Community memberships over time

at any point of time, showing that our approach to prioritize nodes that are not part of any community during the node-community assignment is successful. On the other hand, with the node-community assignment proposed in [SHW17], we saw up to 10% orphan nodes over time. There are significant changes over time, but all values frequently return to the original values. If there are fewer memberships overall, more nodes are just part of a single community and slightly fewer nodes are part of two communities. If there are more memberships overall, the number of nodes that are part of a single community drops and the number of nodes being part of two communities increases, as expected. As the number of memberships decreases again, those numbers quickly recover to their original values as our algorithm reduces over-assignments.

In Figure 4.9, we show the development of the community sizes as well as the total number of communities (right y-axis) over time. As expected, the minimum community size stays stable at 6 nodes. The maximum community size varies significantly, as we could already expect from Figure 4.6. This is due to the large range of community sizes and the relative high exponent of the power law distribution of 2.5. The median community size is at 9, the average is at around 15 and varies over time. The number of communities is varying a lot. This is most likely due to our adjusted event probabilities depending on the number of memberships. As the average community size decreases, we trigger more events that create communities to compensate for the loss of memberships as it can be seen during the first 100 time steps in Figure 4.8. Only around $T = 250$ the number of communities stabilizes, which is around the time when the number of memberships reaches the initial number of memberships again. Without these stabilizing measures, the

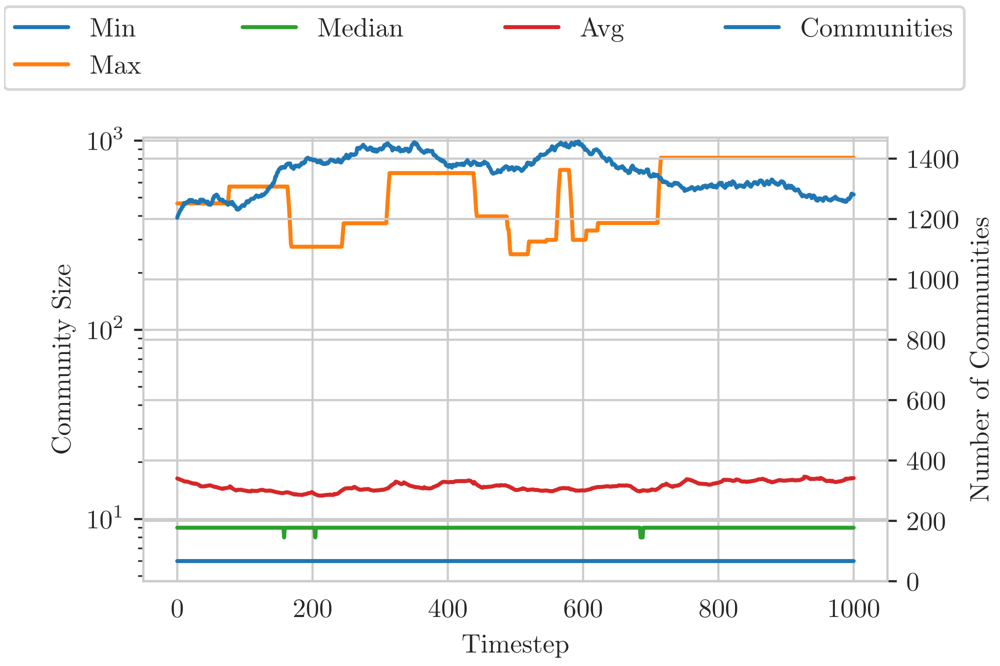


Figure 4.9: Community sizes over time

numbers would be varying much more.

In Figure 4.10, we report the clustering coefficient over time. Compared to the clustering coefficient of around 0.18 reported in [SHW17], the clustering coefficient is higher at around 0.28 as our minimum community size is smaller and it shows a lot more variance as our graph is much more dynamic. A clustering coefficient of 0.28 has also been observed on the LiveJournal social network graph for example [LK14].

4.4.4 Community Detection Algorithm

In this section, we compare how well community detection algorithms can detect the ground truth communities. For this comparison, we use the average weighted F_1 score as introduced in Section 2.3.3.

The most fitting algorithms to evaluate would be the ones that find overlapping communities in a dynamic graph. Unfortunately, we were unable to find implementations of fast enough dynamic overlapping community detection algorithms that are able to detect the communities in our benchmark graphs. In preliminary experiments using the earlier version of our generator [SHW17], we tried a dynamic version of OSLOM [Lan+11] but found that it did not work as intended, producing communities of decreasing quality over time, potentially due to not searching for new candidate communities. In further preliminary experiments, we also tried AFOCS [Ngu+11], but it was only able to process

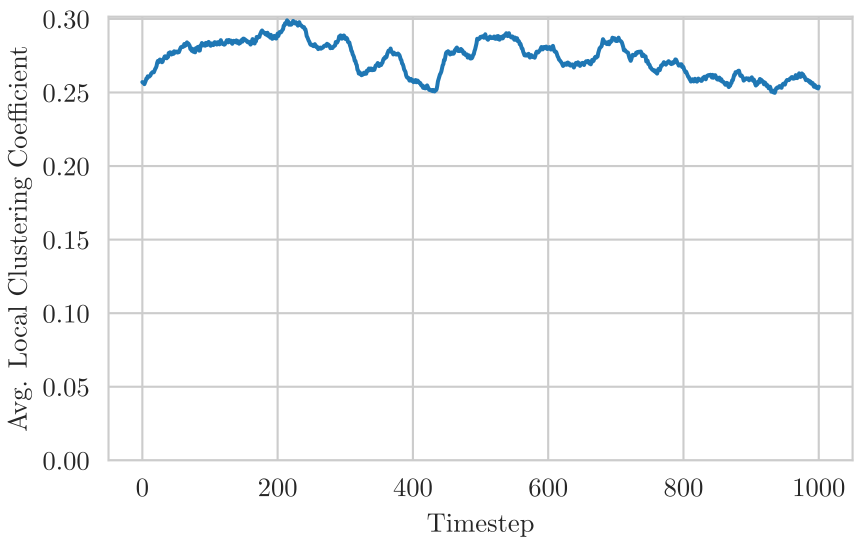


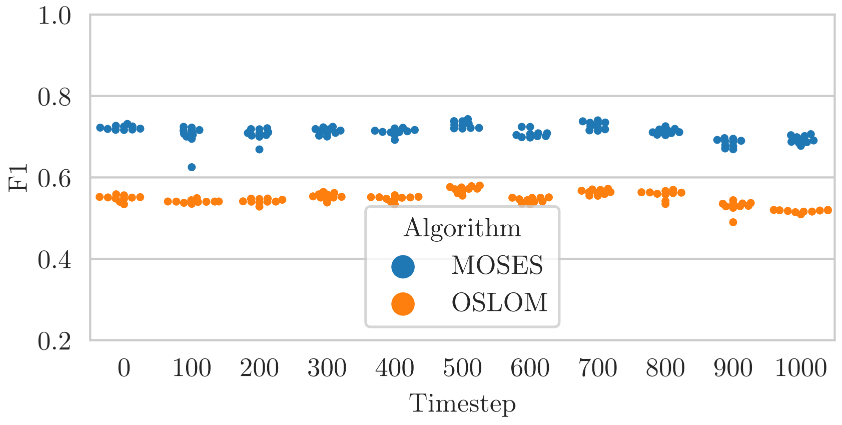
Figure 4.10: Clustering coefficient over time

50 time steps of a graph with 5 thousand nodes within 12 hours and the found communities had average weighted F_1 scores of less than 0.1, i.e., it failed to identify the ground truth community structure.

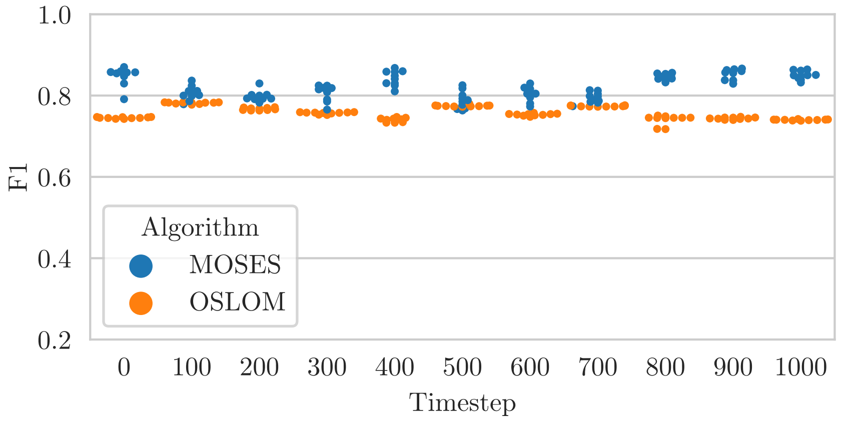
Instead, we evaluate our dynamic generator using the static algorithms MOSES [MH10] and OSLOM [Lan+11], see Section 1.4.5 and 1.4.4 for introductions, as both were shown to have a good performance on the CKB benchmark [Buz+14]. From the stream of events of our generator, we generate snapshots of the graph and the ground truth community structure every 100 time steps. The static community detection algorithms are then run for each individual graph snapshot and the average weighted F_1 score for each of these snapshots is measured independently. As both community detection algorithms are randomized, we run them ten times with different random seeds.

Figure 4.11 shows the performance of the selected algorithms with time and varying values of α and $\epsilon = 2$. As the value of α increases, the intra-community edge probabilities increase, implying that communities are more easily distinguishable. Our experiments confirm the excellent performance of OSLOM, the average F_1 scores are between 0.55 for $\alpha = 1$, almost 0.8 for $\alpha = 2$ and almost 0.9 for $\alpha = 4$. MOSES performs even better for $\alpha = 1$ and $\alpha = 2$, but for $\alpha = 4$, the performance is worse and on similar levels as for $\alpha = 1$. Over the time series, the performance of the algorithms varies, which is expected as the number of memberships and also community sizes varies. However, one cannot see any clear trend and similar scores are repeating, indicating that we can indeed maintain the community structure over time.

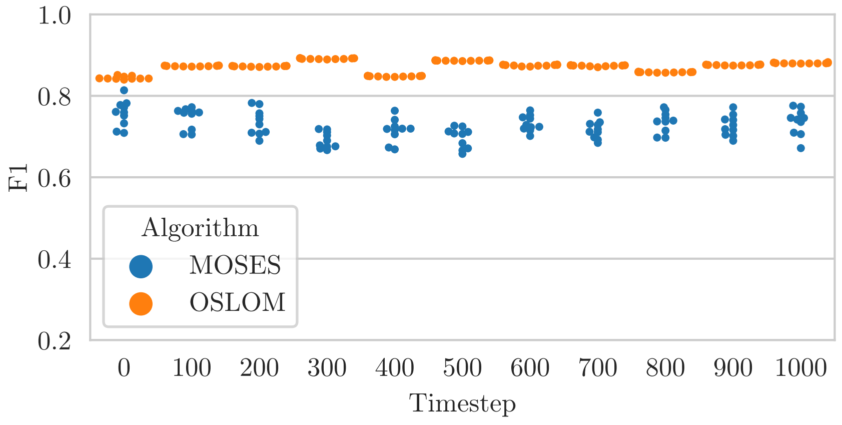
Figure 4.12 shows the performance of the selected algorithms with varying values of ϵ and $\alpha = 2$. It is surprising to see that even extreme values of $\epsilon = 100$ have only a very slight effect on the performance of MOSES. On the other hand, OSLOM performs worse



(a) $\alpha = 1$

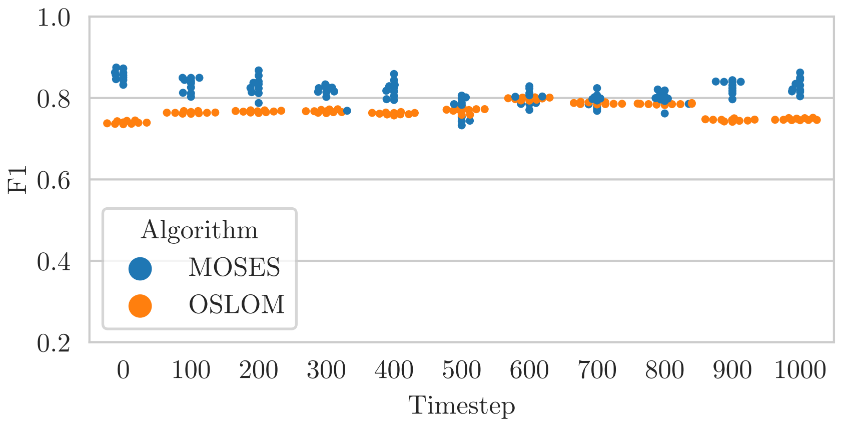


(b) $\alpha = 2$

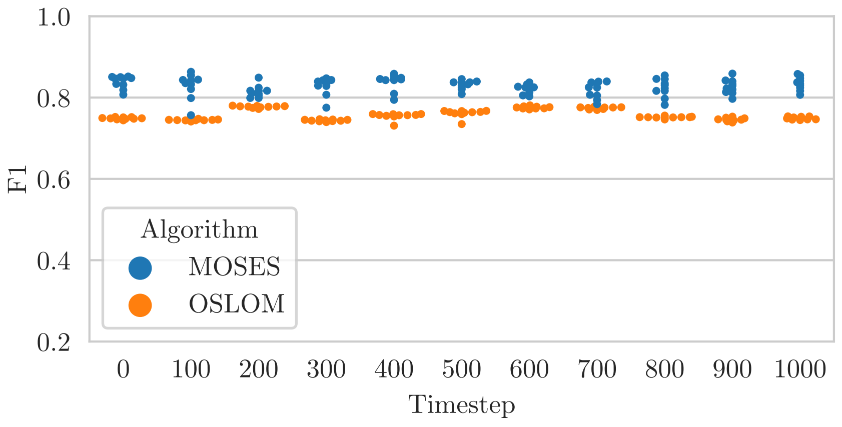


(c) $\alpha = 4$

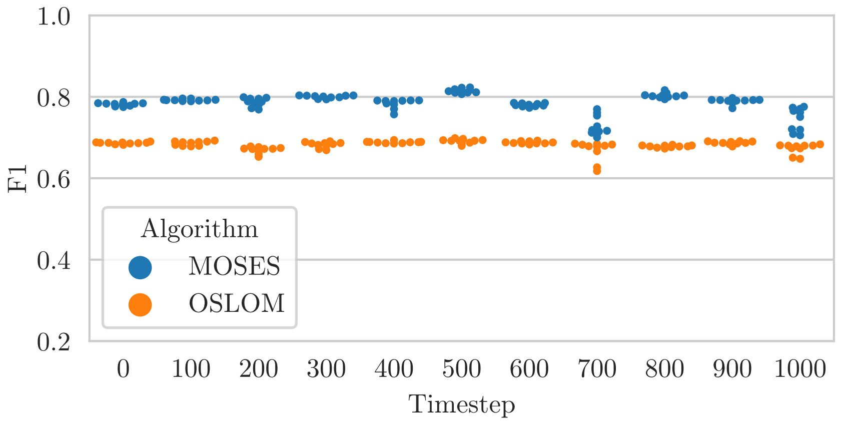
Figure 4.11: Performance of OSLOM and MOSES with varying α and $\epsilon = 2$.



(a) $\epsilon = 1$

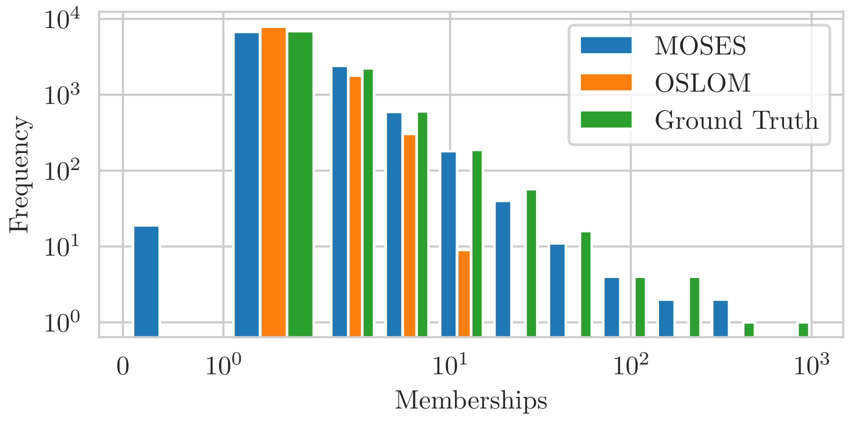


(b) $\epsilon = 10$

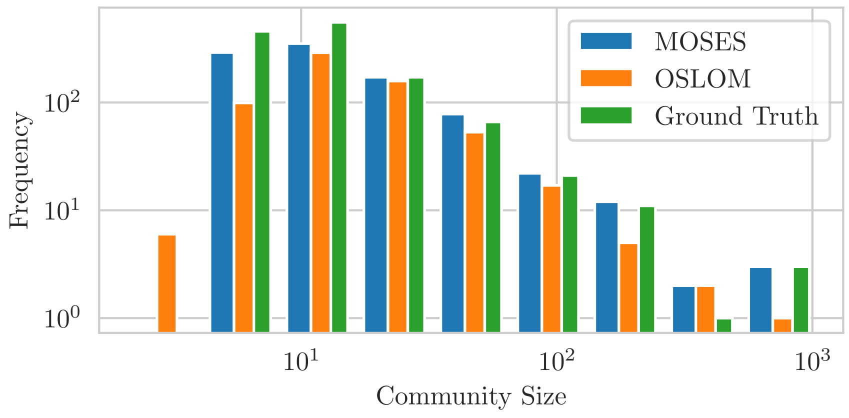


(c) $\epsilon = 100$

Figure 4.12: Performance of OSLOM and MOSES with varying ϵ and $\alpha = 2$.



(a) Community memberships

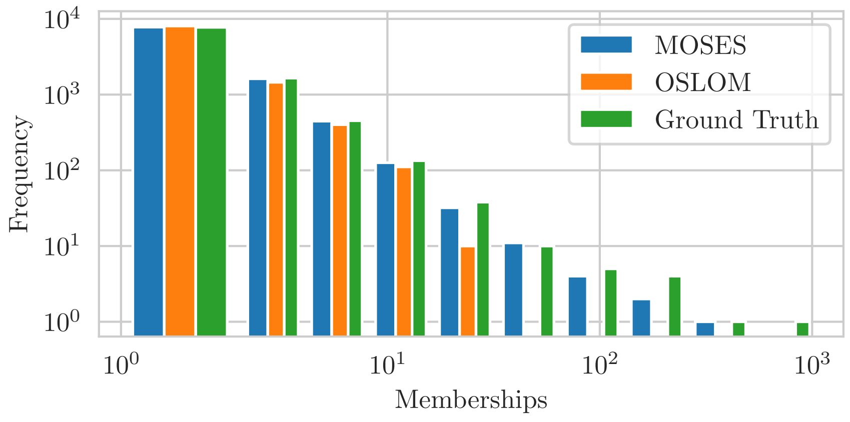


(b) Community sizes

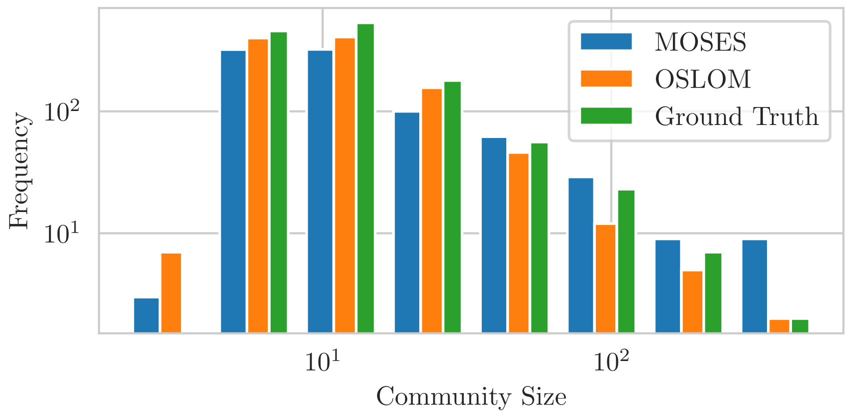
Figure 4.13: Community memberships and sizes for the ground truth and the detected communities for the default parameters of $\alpha = 2$ and $\epsilon = 2$.

with increasing ϵ as expected. The average weighted F_1 score decreases from about 0.8 for $\epsilon = 1$ to 0.7 for $\epsilon = 100$.

To further examine the results of the community detection algorithms, we plot (binned) histograms of the distributions of community memberships and community sizes of both ground truth and detected communities for select values of α and ϵ. As bins, we chose powers of two, i.e., each bin contains all values from the interval $[2^i, 2^{i+1})$ for some $i \in \mathbb{N}$. This allows a direct comparison for each range of community sizes or number of memberships how many communities/nodes should exist and how many were detected. We start with our default configuration of $\alpha = 2$ and $\epsilon = 2$. In Figure 4.13 we see that OSLOM fails to detect nodes that are part of 16 or more communities and thus is unable to identify highly-overlapping nodes. MOSES, on the other hand, is not far off concerning the number of communities per node. For some nodes, though, it does



(a) Community memberships

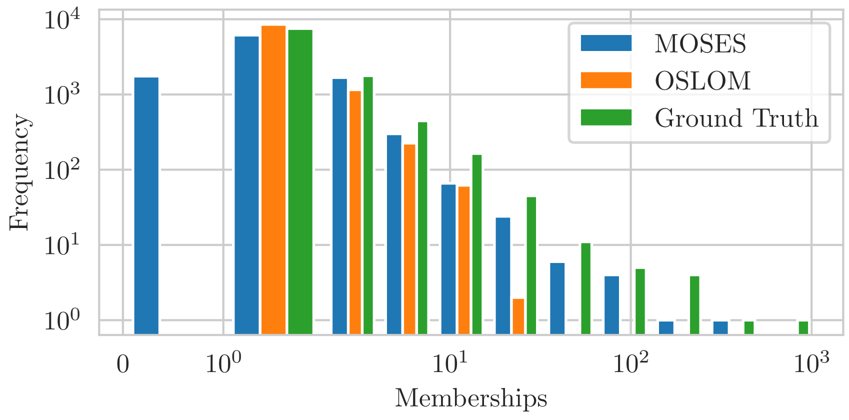


(b) Community sizes

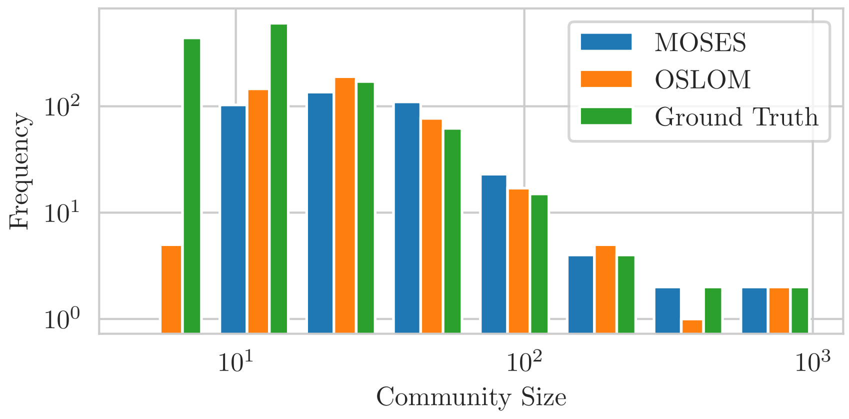
Figure 4.14: Community memberships and sizes for the ground truth and the detected communities for $\alpha = 4$ and $\epsilon = 2$.

not identify any community. Note that in the configuration we used, OSLOM assigns at least one community to every node, otherwise we might see similar effects for OSLOM, too. Concerning the community sizes, in particular OSLOM also detects fewer small communities than there are small communities in the ground truth. MOSES also identifies some communities that are too large, possibly merging several communities into one. Overall, those results confirm the average weighted F_1 scores we saw before where MOSES performs slightly better than OSLOM.

Next, we look at $\alpha = 4$ and $\epsilon = 2$ in Figure 4.14 to identify possible reasons why OSLOM performs better here but MOSES performs worse. Concerning the number of memberships per node, the distributions for MOSES match almost perfectly now, but there are still a couple too few nodes with a high number of memberships. Also, OSLOM is better, it identified about 10 nodes that belong to more than 16 communities. For



(a) Community memberships



(b) Community sizes

Figure 4.15: Community memberships and sizes for the ground truth and the detected communities for $\alpha = 2$ and $\epsilon = 100$.

OSLOM, the distribution of community sizes also matches quite well, though overall there seem to be fewer communities of every size and a few communities smaller than any ground truth community. This also reflects the good F_1 scores of almost 0.9. For MOSES, we see that for all ranges of community sizes above 32 nodes, it detects more communities than the ground truth has. On the other hand, for smaller community sizes, it detects fewer communities than both OSLOM and the ground truth. This suggests that for high values of α, MOSES merges communities even though, as the results for OSLOM show, the communities are actually easier to detect.

As a last example, we look at $\alpha = 2$ and $\epsilon = 100$ in Figure 4.15, i.e., a configuration with extreme background noise. In this configuration, the average weighted F_1 score of MOSES is still at 0.8, while the one of OSLOM is at 0.7. Looking at the distribution of memberships, we can see that for more than 10% of the nodes, MOSES was unable to identify any community. Looking at the distribution of community sizes, we see in particular much fewer small communities. This suggests that small communities whose nodes have a low internal degree basically vanish in the background noise. On the other hand, both algorithms detect more large communities than the ground truth has, suggesting that they might either merge several communities together due to the additional edges in the epsilon community or that it adds unrelated nodes that are connected by edges in the epsilon community to these communities.

All in all, we can conclude that small community sizes pose a significant problem even though we have already increased their minimum size compared to the suggested value of 2 by the original authors of the CKB benchmark [Chy+14]. In particular in small communities with lower values of α, while the average number of edges of a node to its own community might be appropriate, it is quite likely that in some communities there are nodes that have just one or even no edges to their own community. In the LFR model due to the specified minimum degree and fixed mixing parameter, this is not possible. Future research should investigate if such a minimum internal degree might be beneficial for the CKB model, too. Similarly, external degrees might need to be limited, too, to avoid large deviations from the average. To realize this, different models for the graphs that define communities would need to be considered, most likely using the fixed degree sequence model (FDSM, see Section 3.1.1) similar to the LFR benchmark. This would certainly slow down the generation of graphs and make dynamic events more difficult to realize.

4.5 Conclusion

We have introduced the first benchmark graph generator for dynamic overlapping community detection algorithms. The node degrees, the community sizes, and the number of communities per node all follow power law distributions as commonly observed in real-world networks. Compared to our first version of it [SHW17], we made the generator more flexible and the generated graphs more dynamic while ensuring that the properties of the graphs are better preserved over time. In the experimental evaluation, we show that these properties and a realistic local clustering coefficient are not only present in the initial graph but also maintained over time. Further, we demonstrate that our generator is capable of generating graphs with 100,000 nodes and 1000 time steps in about a minute. Our generator is therefore the ideal starting point for an extensive evaluation of existing and novel dynamic community detection algorithms. Further, we show that the existing community detection algorithms MOSES and OSLOM are capable of finding a community structure similar to the ground truth structure. As we show, several parameters allow adjusting how challenging it is. In a detailed analysis, we show that while MOSES tends to detect too large communities, OSLOM has problems with nodes belonging to many communities. An obvious direction for future work is therefore the development of novel algorithms for detecting overlapping communities in dynamic networks. Our detailed analysis of the detected communities suggests that there might be some communities that are not well-connected due to the randomness of the CKB model. A more detailed analysis of such communities and potentially modifying the model to ensure minimum degrees in communities should thus be considered for future work.

Part II

Disjoint Density-Based Communities

5 Introduction to Disjoint Community Detection

The task of disjoint community detection is to divide the input graph G into a disjoint union of communities. In this part, we consider in particular communities that follow the traditional definition of being internally dense and externally sparsely connected. As already discussed in Section 4.1 in the previous chapter, this completely ignores the highly overlapping structure that is observed, e.g., in social networks [YL14] with dense overlaps between communities. Nevertheless, there are several reasons why we consider disjoint, density-based communities an important subject of research. First of all, in some cases overlapping communities might be a better model for a network, but they are too complex for the task. Consider for example the task of understanding huge networks through visualization. Disjoint communities can be visualized by coloring nodes, making it easy to understand which parts of the network belong together. While approaches for visualizing overlapping sets [SAA09; Meu+13] and also overlapping communities [Pei15] exist, they are hardly as easy to grasp as a disjoint structure. In these cases, if there is a disjoint structure that somehow reasonable models the network, we definitely want to be able to detect it. Second, in some cases we might explicitly want a disjoint structure even though it might be a bad model for the network. Consider the related problem of *partitioning* a graph into k blocks of roughly equal size while minimizing the cut between the blocks [BS11]. A successful heuristic is based on first contracting nodes together (similar to the contraction step in the Louvain algorithm, see Section 1.4.1), finding an initial partition on the contracted graph and then refining this initial partition while unpacking the contraction. It has been shown that a disjoint community structure can help the partitioning algorithm with better decisions which nodes not to contract together [MSS14; HS17]. Here, an overlapping community structure would not help as the final partition is non-overlapping, too. Finally, disjoint community detection can be used as a building block for overlapping community detection. For example the Ego-Splitting framework [ELL17] uses a disjoint community detection algorithm to first split the neighborhood of every node into disjoint communities, then creates a copy of each node for each of these communities – so-called personas – and then again detects disjoint communities in an expanded graph of these personas. Thus, scalable disjoint community detection algorithms directly lead to scalable overlapping community detection algorithms.

In this part, we present two very different directions of research towards scalable disjoint community detection. In Chapter 6, we present a community detection algorithm that distributes the data and the computation across a cluster of compute nodes. We show that this algorithm is not only scalable, but that also the quality of the results is sometimes higher than with sequential algorithms. In Chapter 7, we compare different approaches for edge filtering with respect to the goal of obtaining a smaller graph that still yields similar disjoint communities. We show that while some approaches are good at identifying edges inside communities, the filtered graphs yield different communities. Filtering random edges better maintains the structure, on the tested graphs we can remove 40 to 60% of the edges and still obtain similar communities. While our distributed community detection algorithm yields the more scalable approach, being able to detect communities on a twice as large graph without needing more RAM or a compute cluster might make a difference in practice, too. Further, studying different ways to determine the importance of an edge is interesting not only for filtering, but also for prioritizing edges. In Chapter 11 on local community detection, we show that prioritizing certain edges helps to expand a community around a seed node.

6 Distributed Graph Clustering using Modularity and Map Equation

This chapter is based on joint work with Ben Strasser, Dorothea Wagner and Tim Zeitz [Ham+18a]. Compared to the publication, we re-added some parts that were omitted due to space constraints and reference parts of this thesis where appropriate. Further, we added more recent related work and a comparison to it based on published results.

When dealing with data that does not fit into the RAM of a single machine, a natural approach is to instead distribute the data across a compute cluster. We therefore study distributed extensions of established single machine clustering algorithms. In this chapter, we present two versions of a simple distributed graph clustering algorithms. They compute a disjoint clustering by optimizing the quality measures modularity [NG04] and map equation [RAB09], respectively. The algorithms for the two measures, DSLM-Mod and DSLM-Map, differ only slightly. Adapting them for similar quality measures is straightforward. They are based on Thrill [Bin+16], a distributed big data processing framework that implements an extended MapReduce [DG08] model. We conduct an extensive experimental study on real-world graphs and on synthetic benchmark graphs with up to 68 billion edges. Our algorithms are fast while detecting clusterings similar to those detected by other sequential, parallel and distributed clustering algorithms. Compared to the distributed GossipMap [BH15] algorithm, DSLM-Map needs less memory, is up to an order of magnitude faster and achieves better quality.

6.1 Related Work

Existing distributed approaches follow one of two approaches.

The first is to partition the graph into a subgraph per machine. Each subgraph is then clustered independently on one machine. Then, all nodes of each cluster are merged summing up weights of parallel edges. The resulting coarser graph is clustered on a single machine. This assumes the coarsened graph fits in the memory of a single machine. In [ZY16], for the partitioning, the input node ID range is chunked into equally sized parts. This can work well, but is problematic if input node IDs do not reflect the graph structure. In [Wic+14], the input graph is partitioned using the non-distributed, parallel graph partitioning algorithm ParMETIS [KK98]. While this is independent of node IDs, it requires that the graph fits into the memory of one machine for the partitioning.

The second approach consists of distributing the clustering algorithm itself. Using MPI, [Que+15] have introduced a distributed extension of the Louvain algorithm [Blo+08].

Similarly, [Lin+16] have presented an algorithm that uses the GraphX framework. Another algorithm named GossipMap is presented in [BH15] which uses the GraphLab framework. InfoFlow [Fun19] is based on Apache Spark. Our algorithms also use this second approach.

In later works [Gho+18b; ZY18; FA19], a combination of both approaches has been proposed: while the graph is partitioned, information about neighboring nodes is exchanged from time to time. Therefore, the quality still depends on the partition, but possibly not that heavily.

While many of these related algorithms heuristically optimize modularity, GossipMap, InfoFlow and the last two approaches [Fun19; ZY18; FA19] optimize the map equation. Other community detection formalizations have been considered. For example, EgoLP [Buz+14] is a distributed algorithm to find overlapping clusters.

6.2 Our Contribution

We propose two distributed graph clustering algorithms, DSLM-Mod and DSLM-Map, that optimize modularity and map equation, respectively. Our algorithms are the first graph clustering algorithms based on Thrill [Bin+16], a distributed big data processing framework written in C++ that implements an extended MapReduce [DG08] model. Our algorithms are easy to extend for optimizing different density-based quality measures. To evaluate the clustering quality, we compare against ground truth communities on synthetic LFR [LFR08] graph clustering benchmark graphs with up to 68 billion edges. Even for these graphs, 32 hosts of a compute cluster are enough. Our results show that our algorithms scale well and DSLM-Map is better at recovering the ground truth than the sequential Infomap algorithm [RAB09]. On real-world graphs, our algorithms perform similarly to non-distributed baseline algorithms in terms of the quality of the detected clusterings and stability between different runs. We evaluate both similarities and quality scores, as for quality scores small changes can result in vastly different clusterings [GMC10].

Similar to most related work, we make implicit assumptions on the structure of the graph. We assume that all edges incident to nodes in a single cluster fit into the memory of a single compute node. In practice, this is only a limitation when a graph has huge clusters. In many scenarios like social networks or web graphs, this is no limitation as cluster sizes are not that huge. Our algorithms can be modified to avoid these restrictions, but this would increase running times.

Outline. In the following we introduce our notation and present the quality measures we optimize. We also give a brief introduction to Thrill. In Section 6.4, we present our algorithms. In Section 6.5 we present our experimental results. We conclude in Section 6.6.

6.3 Preliminaries

We use the notation introduced in Section 1.2. While all our input graphs are unweighted, intermediate steps of our algorithms use weighted graphs. Therefore, we treat all graphs as weighted. In the context of this chapter, a clustering $\mathcal{C}$ is a disjoint set of clusters such that every node is part of exactly one cluster.

6.3.1 Thrill

Thrill [Bin+16] is a distributed C++ big data processing framework. It can distribute the program execution over multiple machines and threads within a machine. Each thread is called *worker*. Every worker executes the same program.

If there is not enough main memory available, Thrill can use local storage such as an SSD as external memory to store parts of the processed data. This makes it possible to work with datasets larger than the combined main memory of all hosts.

Data is maintained in distributed immutable arrays (DIA). Distributed Thrill operations are applied to the DIAs. For example, Thrill contains a *sort* operation, whose input is a DIA and whose output is a new sorted DIA. Similarly, the *zip* operation combines two DIAs of the same length into one DIA where each element is a pair of the two original elements.

Thrill also supports DIAs with elements of non-uniform size, as long as each element fits into the memory of a worker. This allows elements to be arrays.

Apart from zip and sort, we use the following operations: The *map* operation applies a function to each element of a DIA. The return values are the elements of the new DIA. *Flatmap* is similar, but the function may emit 0, 1, or more elements. This is similar to the map operation in the original MapReduce model [DG08].

A DIA can be *aggregated* by a key component. All elements with the same key are combined and put into an array. This is similar to the reduce operation in the original MapReduce model [DG08]. An aggregation is much more efficient if the keys are consecutive integers. In that case, the result is also automatically sorted by the keys. We use this optimized variant for all aggregations that are based on node IDs.

6.4 Algorithm

The basis of our algorithm is the Louvain algorithm [Blo+08], a fast algorithm for modularity optimization that delivers high-quality results. The original Infomap algorithm proposed for optimizing the map equation [RAB09] is based on the same scheme, but introduces additional steps to improve the quality. For an introduction to modularity and map equation and formal definitions of both, we refer to Section 1.3.3 and 1.3.4. While we have introduced the Louvain algorithm already in Section 1.4.1, we repeat it here to establish the notation that we use in the following sections to describe our distributed algorithm.

Initially, every node is in its own cluster. This is called a singleton clustering.

In the local moving phase, the Louvain algorithm works in rounds. In each round, it iterates in a random order over all nodes v. For every node v, it considers v's current cluster and all clusters C such that there is an edge from v to a node of C. For all these clusters, the difference in quality score $\Delta_{v,C}$ if v was to be moved into C is computed. If an improvement is possible, v is moved into a cluster with maximal $\Delta_{v,C}$, resolving ties uniformly at random. The local moving phase stops when no node is moved in a round or a maximum number of rounds is reached.

After the local moving phase, the contraction phase starts. All nodes inside a cluster are merged into a single node. The weights of all edges from a cluster C to D are summed up and added as an edge from the node representing C to the node representing D. All edge weights within a cluster are summed up and added as a loop edge. The contraction does not change the quality score. On the contracted graph, the algorithm is applied recursively. It terminates when the contraction phase is entered and the clustering is still the singleton clustering.

6.4.1 Calculation of Scores

To evaluate if a node v shall be moved from its cluster C to a cluster D, we need to know the difference in the quality measure $\Delta_{v,D}$. For the calculation of $\Delta_{v,D}$, let C^- denote the cluster C without node v and C^+ the cluster C with node v. Both modularity and map equation allow computing $\Delta_{v,D}$ using just information about C and D and, for map equation, how $\mathrm{cut}_w(\mathcal{C})$ changes. The latter can also be obtained easily if we know the value before the move. This suffices as most terms in the modularity and the map equation formula are just sums over all clusters. When moving a node, only the parts for the affected clusters C and D change. Thus, all other terms cancel out when calculating the difference. Only the first term in the map equation is different as it is a sum of all cut values. However, this sum can be quickly updated, too. For modularity, we obtain the following formula:

$$\Delta\mathcal{Q}_{v,D} := 2 \cdot \left(\frac{\mathrm{cut}_w(v, D^-) - \mathrm{cut}_w(v, C^-)}{\mathrm{vol}_w(V)} - \frac{\deg_w(v)}{\mathrm{vol}_w(V)} \cdot \frac{\mathrm{vol}_w(D^-) - \mathrm{vol}_w(C^-)}{\mathrm{vol}_w(V)} \right)$$

Since the absolute difference is not required to select the best cluster for a given node, this formula can be simplified further. Specifically, if the set of possible clusters is guaranteed to contain the current cluster, we can drop all parts in the formula referring to the current cluster C. This will allow optimizations later on.

For the map equation, this formula contains more terms. Due to the logarithms fewer terms cancel out:

$$\begin{aligned}
\Delta L_{v,D} := {} & \mathrm{plogp}\left(\frac{\mathrm{cut}_w(\mathcal{C}) + \mathrm{cut}_w(v, C^-) - \mathrm{cut}_w(v, D^-)}{\mathrm{vol}_w(V)}\right) - \mathrm{plogp}\left(\frac{\mathrm{cut}_w(\mathcal{C})}{\mathrm{vol}_w(V)}\right) \\
& - 2\,\mathrm{plogp}\left(\frac{\mathrm{cut}_w(C^-)}{\mathrm{vol}_w(V)}\right) + \mathrm{plogp}\left(\frac{\mathrm{cut}_w(C^-) + \mathrm{vol}_w(C^-)}{\mathrm{vol}_w(V)}\right) + 2\,\mathrm{plogp}\left(\frac{\mathrm{cut}_w(C^+)}{\mathrm{vol}_w(V)}\right) \\
& - \mathrm{plogp}\left(\frac{\mathrm{cut}_w(C^+) + \mathrm{vol}_w(C^+)}{\mathrm{vol}_w(V)}\right) - 2\,\mathrm{plogp}\left(\frac{\mathrm{cut}_w(D^+)}{\mathrm{vol}_w(V)}\right) + 2\,\mathrm{plogp}\left(\frac{\mathrm{cut}_w(D^-)}{\mathrm{vol}_w(V)}\right) \\
& + \mathrm{plogp}\left(\frac{\mathrm{cut}_w(D^+) + \mathrm{vol}_w(D^+)}{\mathrm{vol}_w(V)}\right) - \mathrm{plogp}\left(\frac{\mathrm{cut}_w(D^-) + \mathrm{vol}_w(D^-)}{\mathrm{vol}_w(V)}\right)
\end{aligned}$$

For obtaining the updated volume and cut values of C and D, we only need to know the previous cut and volume values of C and D, the degree of v and the cut of v with C^- and D^-. The latter can easily be obtained by iterating over the neighbors of v. In the sequential algorithms, the cut and volume of each cluster as well as the global cut are cached and updated after every move.

6.4.2 Distributed Synchronous Local Moving (DSLM)

In local moving, the i-th move depends on the clustering after the first $i - 1$ moves are executed. This data dependency makes the parallelization difficult. We therefore split a round into sub-rounds. Every node picks a random sub-round in which it is *active*. In the i-th sub-round, all active nodes are moved synchronously and in parallel with respect to the clustering after the $(i - 1)$-th sub-round. For the first sub-round, this is with respect to the initial clustering. We call this scheme *synchronous local moving*. For our *distributed* synchronous local moving, a global hash function h maps a tuple of a node ID v, the number of completed rounds and a global seed onto the sub-round r_v in which v is active. Figure 6.1 illustrates the data flow of our algorithm.

We represent a graph and its clustering as two DIAs. They have length n and are stored sorted by their first item, the node ID v. The graph DIA stores triples $(v, \langle u_i \rangle, \langle w_i \rangle)$, where $\langle u_i \rangle$ and $\langle w_i \rangle$ are equally-sized arrays. For every i, there is an edge v, u_i with weight w_i. We store every edge twice such that it is accessible from both incident nodes. The clustering DIA of pairs (v, C) stores for every node v its cluster ID C.

In DSLM, a sub-round is composed of a bidding and a compare step. In the bidding step, the clusters place bids for active nodes. In the compare step, every active node compares its bids and becomes part of the cluster with the best bid.

To allow a node v to compute the change in modularity or map equation when joining the neighboring cluster C, each bid contains: (a) volume $\text{vol}_w(C \setminus v)$, (b) cut weight $\text{cut}_w(v, C \setminus v)$ between $C \setminus v$ and v, and (c) cut weight $\text{cut}_w(C)$ between C and the remaining graph.

The bidding step starts by zipping the clustering DIA and graph DIA and aggregating this zipped DIA by cluster ID. The result is a DIA with one large element per cluster C. Each element contains all nodes in the cluster C and the neighborhoods of these nodes. This allows us to compute the measures (a), (b), and (c) using a non-distributed algorithm inside a worker. Using a flatmap operation, our algorithm emits for every cluster C bids for all active nodes inside C and adjacent to C. It can determine which nodes are active as the hash function h is globally known. The generated bid DIA consists of quintuples $(C, v, \text{vol}_w(C \setminus v), \text{cut}_w(v, C \setminus v), \text{cut}_w(C \setminus v))$. Each quintuple is a bid of cluster C for active node v.

The compare step aggregates the bid DIA by node v. After the aggregation, the result is zipped with the graph DIA to obtain the nodes' degree and loop edge weight. In a map operation, we use this information to compute for every active node the best bid and return the according cluster. We retrieve the old cluster ID for non-active nodes by zipping with the original clustering DIA. This yields the updated clustering DIA, which is the input of the next sub-round.

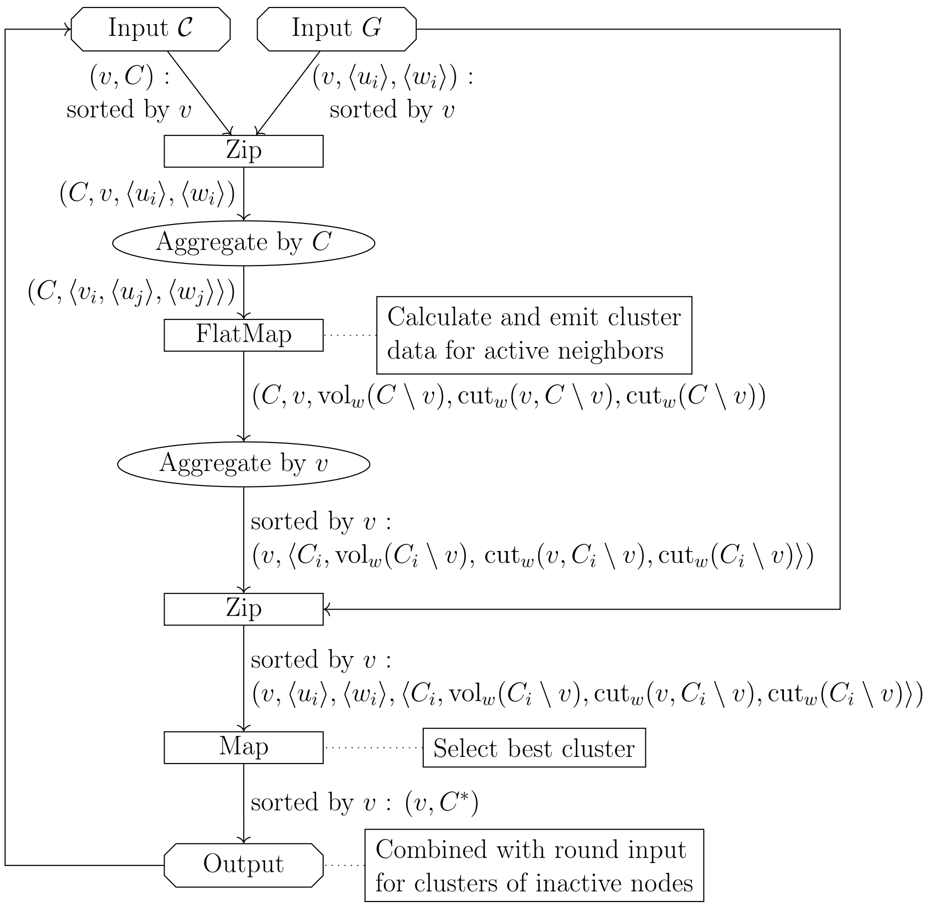

Input $\mathcal{C}$
Input G
(v, C) : sorted by v
$(v, \langle u_i \rangle, \langle w_i \rangle)$: sorted by v
Zip
$(C, v, \langle u_i \rangle, \langle w_i \rangle)$
Aggregate by C
$(C, \langle v_i, \langle u_j \rangle, \langle w_j \rangle \rangle)$
FlatMap
Calculate and emit cluster data for active neighbors
$(C, v, \mathrm{vol}_w(C \setminus v), \mathrm{cut}_w(v, C \setminus v), \mathrm{cut}_w(C \setminus v))$
Aggregate by v
sorted by v : $(v, \langle C_i, \mathrm{vol}_w(C_i \setminus v), \mathrm{cut}_w(v, C_i \setminus v), \mathrm{cut}_w(C_i \setminus v) \rangle)$
Zip
sorted by v : $(v, \langle u_i \rangle, \langle w_i \rangle, \langle C_i, \mathrm{vol}_w(C_i \setminus v), \mathrm{cut}_w(v, C_i \setminus v), \mathrm{cut}_w(C_i \setminus v) \rangle)$
Map
Select best cluster
sorted by v : (v, C^*)
Output
Combined with round input for clusters of inactive nodes

Figure 6.1: DSLM data flow

Implementation Details and Optimizations. If modularity is optimized, our algorithm can be improved in two ways. First, we can omit $\mathrm{cut}_w(C \setminus v)$ as it is only needed for the map equation. Second, as mentioned above, we can compare bids without knowing the current cluster. This allows us to use a pairwise reduction instead of one that first waits for all elements. As we still need the node's degree, each worker stores the degree of the nodes that are reduced on that worker in a plain array. This is possible because we know on which worker each node will end up.

6.4.3 Distributed Contraction and Unpacking

The contraction is performed in three steps: (a) obtain consecutive cluster IDs, (b) replace all node IDs by the cluster IDs, and (c) combine multi-edges.

We first zip the graph and clustering DIAs and aggregate them by cluster ID. To get consecutive cluster IDs, we replace them with the element positions. From the result, which contains tuples $(C, \langle v_i, \langle u_j \rangle, \langle w_j \rangle \rangle)$, we derive two DIAs.

The first DIA is a new clustering DIA with consecutive cluster IDs. To obtain it, we first drop the neighborhood information. We store this intermediate DIA $(C, \langle v_i \rangle)$ for the unpacking phase. Then, we expand it into pairs (v_i, C) of node and cluster ID using a flatmap operation and sort by node IDs.

The second DIA is the contracted graph DIA. To obtain it, we first emit a triple (C_u, v, w) for every node v that is a neighbor of a node in C_u in a flatmap operation. The weight w is the sum of all edge weights from nodes in C_u to v. We aggregate this DIA by v, zip it with the new clustering DIA and replace v by v's cluster ID C_v. We then aggregate by C_v to obtain pairs $(C_v, \langle C_{u,i}, w_i \rangle)$ containing the neighboring clusters $C_{u,i}$ for every C_v. Finally, we sum up the weights w_i for the same neighboring cluster for every cluster C_v in a map operation.

To unpack the clustering calculated in a level, we zip the clustering DIA (v, C_v) with the intermediate clustering DIA $(v, \langle v_i \rangle)$ of a cluster v and its nodes $\langle v_i \rangle$ from the previous contraction phase. A flatmap operation assigns the cluster ID C_v of the contracted node to all original nodes $u \in \langle v_i \rangle$, resulting in a clustering DIA (u, C_u). After sorting it by node, it is returned to the next level.

6.5 Experiments

In this section, we present an experimental evaluation of our algorithm DSLM[1]. The source code of our implementation is publicly available on GitHub[2]. We first describe our experimental setup. Then, we present weak scaling experiments to evaluate the running time, compare the quality on LFR benchmark graphs [LFR08] and evaluate the performance on established real-world benchmark data.

All running time experiments were performed on a compute cluster. Each compute node has two 14-core Intel Xeon E5-2660 v4 processors (Broadwell) with a default frequency of 2 GHz, 128 GiB RAM and 480 GiB SSD. They are connected by an InfiniBand 4X FDR Interconnect. We use the TCP back-end of Thrill due to problems with the combination of multithreading and OpenMPI. We use Thrill's default parameters, except for the block size, which determines the size of data packages sent between the hosts. Preliminary experiments found that a block size of 128 KiB instead of the default 2 MiB yields the best results.

For our algorithms, we use four sub-rounds as suggested in a preliminary study [Zei17]. Using fewer results in problems with the convergence. Using more does not significantly improve quality but increases running time. In each local moving phase, we perform at most eight rounds. All experiments were performed with 16 threads per host. More threads do not improve the running times much further. Preliminary experiments indicate that the performance is RAM bound.

Apart from DSLM-Mod and DSLM-Map that optimize modularity and map equation, we also evaluate a variant DSLM-Mod w/o Cont. that stops after the first local moving

[1]This section only covers parts of our experiments. Under `https://github.com/kit-algo/distributed_clustering_thrill_evaluation` you can find additional analyses, links to our raw data and information on how to explore our data on your own.

[2]`https://github.com/kit-algo/distributed_clustering_thrill`

phase. This significantly decreases the running time and surprisingly also improves the quality on synthetic instances. We evaluate this behavior in more detail in Section 6.5.2.

For modularity, we compare against our own implementation of the sequential Louvain algorithm [Blo+08] and the shared-memory parallel PLM [SM16]. For map equation, we compare against the sequential Infomap [RAB09], the shared-memory parallel RelaxMap [Bae+17] and the distributed GossipMap [BH15] implementations.

In a preprocessing step, we remove degree zero nodes, make the ID space consecutive and randomize the node order. This ensures that our algorithms are independent of input order and improves load balancing.

All experiments were repeated 10 times with different random seeds. We report averaged results and standard deviation where possible as error bars.

To compare clusterings, either with ground truth communities or with a clustering calculated using a different algorithm, we use the *adjusted rand index* (ARI) [HA85], see Section 2.3.2 for an introduction.

During the experiments, the meltdown and spectre vulnerabilities became public and performance impacting patches were applied to the machines. Rerunning some experiments showed a slowdown of up to 1.6 for runs with 32 hosts but no significant slowdown for runs with a single host. We did not have the resources to rerun all experiments. Also, we expect the performance of the machines to change further in the future. Patches with less impact (Retpoline) are available but have not been rolled out yet. More vulnerabilities have been discovered in the meantime and it is unclear if fixes for them will have further performance implications[3]. At the point of initial patch distribution, most distributed algorithm runs were already done. About half of the GossipMap runs on the real world graphs were performed afterwards and are excluded from the running time reports. All runs for non-distributed algorithms were performed with patches applied, as their performance should not have been affected significantly.

Synthetic Instance Generation. Our synthetic test data is generated using the established LFR benchmark generation scheme [LFR08] that we have described in Section 2.1.1. To generate graphs of up to 512 million nodes and 67.6 billion (undirected) edges in a reasonable time, we use the external memory LFR generator implementation [Ham+18b] described in Chapter 3.

We set a minimum degree of 50 and a maximum degree of 10 000 with a power law exponent of 2. This leads to an average degree of approximately 264. For the communities, we set 50 as minimum and 12 000 as maximum size with a power law exponent of 1. This corresponds to the **const** parameter set in Section 3.10. As explained there, these settings are inspired by the degree distribution of the Facebook network in May 2011 with 721 million active users [Uga+11]. Unless otherwise noted, we set the mixing parameter μ to 0.4. This means that in contrast to previous experiments on the behavior of clustering algorithms on larger LFR benchmark instances [Emm+16], our cluster sizes are not scaled as we increase the graph size and the diameter of the clusters remains small. This avoids the field of view limit experienced in previous work [Emm+16].

[3] `https://securityaffairs.co/wordpress/72158/hacking/spectre-ng-vulnerabilities.html`

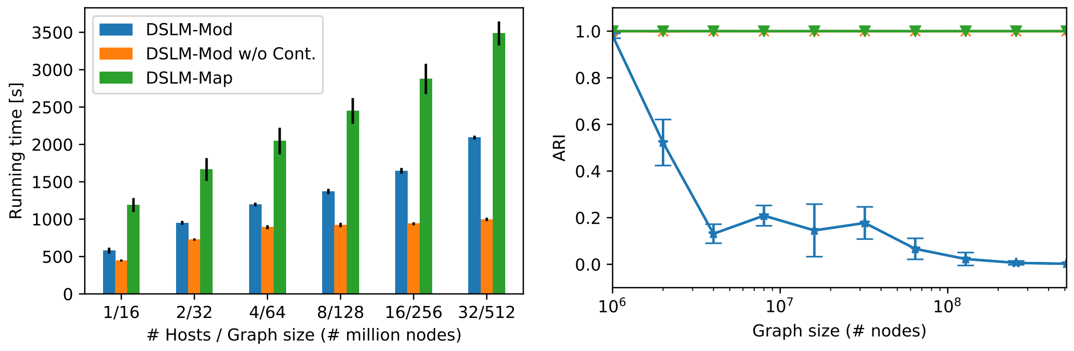


Figure 6.2: Weak scaling: running time of our distributed algorithms and ARI with ground truth. The DSLM-Mod w/o Cont. ARI line is hidden by the DSLM-Map line.

6.5.1 Weak Scaling

For the weak scaling experiments, we use LFR graphs with 16, 32, 64, 128, 256 and 512 million nodes. We cluster them on 1, 2, 4, 8, 16, and 32 hosts respectively. The left part of Figure 6.2 shows the running time of our algorithms. Our algorithms utilize almost the entire available RAM. GossipMap is less memory-efficient and was unable to cluster the graphs in these configurations and crashed.

With a linear time algorithm and perfect scaling, we would expect that the running time remains constant as we increase graph size and the number of nodes. For the variant of DSLM-Mod w/o Cont., the running time actually does not increase much. The running time of the full DSLM-Mod and DSLM-Map algorithms increases approximately linearly though as the number of hosts is scaled exponentially. The reason for this is that LFR graphs get very dense during contraction and thus in particular larger graphs still have a significant amount of edges after the contraction. Also, DSLM-Map is approximately a factor of two slower than DSLM-Mod. This is expected as the optimizations described at the end of Section 6.4.2 are not applicable to DSLM-Map.

6.5.2 Quality

First, we evaluate the quality of the clusterings obtained in the weak scaling experiment. The right part of Figure 6.2 depicts the similarities of the clusterings found by our algorithms and the ground truth. From the plot, we observe that DSLM-Map finds a clustering very close to the ground truth. DSLM-Mod w/o Cont. achieves similar results. Unfortunately, DSLM-Mod fails to find a clustering similar to the ground truth on the larger instances. This shows that after the contraction, clusters are merged that should not be merged. To verify if the worse results of DSLM-Mod are due to the resolution limit, we started a sequential Louvain algorithm on a graph where we contracted the ground truth. This algorithm indeed merges clusters, showing that the resolution limit is relevant here. However, the thereby detected clusters are much more similar to the ground truth than those detected by DSLM-Mod and even the ground truth alone has

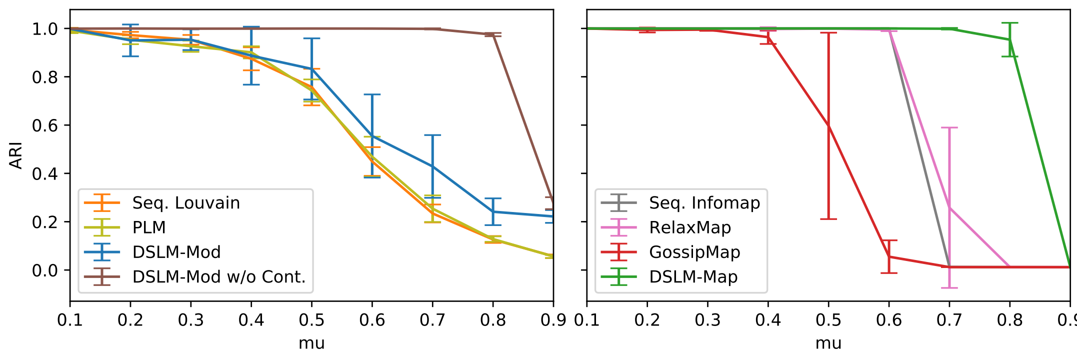


Figure 6.3: Adjusted rand index with ground truth for $\mu \in [0.1, 0.9]$.

higher modularity scores than those found by DSLM-Mod.

We also use smaller LFR graphs with 1M nodes and varying mixing parameter to compare the quality of the communities found by all compared algorithms. Figure 6.3 shows the adjusted rand index of the detected clusterings with the ground truth. DSLM-Mod w/o Cont. and DSLM-Map outperform all other algorithms by a significant margin. On average, DSLM-Mod still outperforms the other modularity-optimizing algorithms. For all values of μ, the ground truth has a higher modularity score than the clustering found by the modularity-optimizing algorithms. Merging clusters of the ground truth again improves the modularity score but leads to clusterings that still have an ARI of above 0.99 for $\mu < 0.9$. With the algorithms optimizing map equation, the situation is similar. For $\mu < 0.8$, the ground truth, which DSLM-Map consistently finds, has a better map equation score than the clusterings found by all other algorithms. For $\mu \geq 0.8$, a singleton clustering yields a better map equation score than the ground truth clustering. GossipMap finds neither good map equation scores nor the ground truth for $\mu > 0.4$.

Overall, for these LFR benchmark graphs, DSLM seems to be superior to sequential local moving. Examining sequential local moving algorithms, we noticed that high-degree nodes attract many nodes in the first local moving round. After a few nodes join their cluster, many others follow. In contrast to that, with DLSM, 25% of the nodes can join the cluster of another node before any cluster sizes come into play. Apparently, this avoids such accumulation effects.

6.5.3 Real-World Graphs

To assess whether our results on LFR benchmark graphs are also true for real-world graphs, we performed experiments on a set of different real-world graphs. From the Stanford Large Network Dataset Collection [LK14], we include three social networks (com-LiveJournal, com-Orkut and com-Friendster), see Section 2.2.2 for an introduction. From the 10th DIMACS Implementation challenge [Bad+14], we include two web graphs where nodes represent URLs and edges links between them (uk-2002 and uk-2007-05, see also Section 2.2.1).

Table 6.1: Average running time in seconds of the algorithms on the real-world graphs.

	# Nodes	# Edges	# Hosts	Louvain	PLM	DSLM-Mod	DSLM-Mod w/o Cont.	Infomap	RelaxMap	GossipMap	DSLM-Map
LiveJournal	4M	34M	8	99	25	31	14	1329	163	372	49
Orkut	3M	117M	8	170	53	47	34	2405	415	700	84
uk-2002	18M	261M	8	572	142	46	22	6656	240	682	52
Friendster	66M	1806M	16	6002	1755	1047	742	oom	oom	13743	1161
uk-2007-05	105M	3302M	16	7993	2520	151	106	oom	oom	4211	214

We clustered these graphs both with the sequential baseline algorithms and our distributed algorithms. Table 6.1 depicts the sizes of the graphs, the number of hosts we used for the distributed algorithms and the running times in seconds. RelaxMap and GossipMap use the directed version of the map equation that includes a PageRank approximation as preprocessing step. To allow for a fair comparison, we only report the running time of the actual clustering step after this preprocessing. As three GossipMap runs on uk-2007-05 crashed, there are less samples. Unfortunately, both the original Infomap implementation and RelaxMap were not able to cluster all instances. On the two largest graphs, 128 GB of RAM were not enough memory (oom).

With 8 or 16 hosts, our distributed algorithms are almost always faster than the sequential and shared-memory parallel algorithms. Note that due to the randomized node order, PLM is slower in our experiments than reported in [SM16]. DSLM-Map with 8 hosts is more than a factor of 5, for uk-2002 even a factor of 20 faster than RelaxMap and also a factor of 10 faster than GossipMap. DSLM-Mod is faster than DSLM-Map, but the difference is less pronounced than in the weak scaling experiments. This shows the advantage of our algorithmic scheme in combination with the efficient Thrill framework.

Table 6.2 shows the average modularity and map equation scores obtained by the algorithms. We observe that PLM on average finds the best modularity scores with a minor exception on uk-2002 where Louvain finds better values. DSLM-Mod performs slightly worse, DSLM-Mod w/o Cont. significantly worse. This shows that DSLM-Mod w/o Cont., which performed really well on the LFR benchmark graphs, is unsuited for real-world graphs.

The best map equation scores are found by the Infomap algorithm where it finished the computation. Since RelaxMap and GossipMap use the directed map equation, we also include the directed Infomap algorithm to evaluate if using the directed map equations leads to different results. Surprisingly, in some cases Infomap optimizing the directed map equation finds better clusterings with respect to the undirected map equation than the undirected Infomap, though the differences are very small. On the two smallest graphs, RelaxMap finds better clusterings than the distributed algorithms. On uk-2002,

Table 6.2: Average modularity/undirected map equation scores obtained by the respective algorithms.

	Louvain	PLM	DSLM-Mod	DSLM-Mod w/o Cont.	Infomap	Directed Infomap	RelaxMap	GossipMap	DSLM-Map
LiveJournal	0.752	**0.752**	0.749	0.591	**9.899**	9.900	9.943	9.963	9.981
Orkut	0.664	**0.666**	0.658	0.524	11.826	**11.825**	11.849	11.979	11.896
uk-2002	**0.990**	0.990	0.990	0.879	6.458	**6.458**	6.476	6.550	6.468
Friendster	0.622	**0.627**	0.616	0.553	oom	oom	oom	16.271	**14.785**
uk-2007-05	0.996	**0.996**	0.996	0.919	oom	oom	oom	9.034	**8.057**

Table 6.3: Average similarities in terms of ARI with best clustering found according to the respective quality score. Underlined entries indicate the algorithm which found the clustering with the best score.

	Louvain	PLM	DSLM-Mod	DSLM-Mod w/o Cont.	Infomap	Directed Infomap	RelaxMap	GossipMap	DSLM-Map
LiveJournal	0.571	0.639	0.600	0.179	0.976	0.973	0.376	0.784	0.769
Orkut	0.632	0.625	0.659	0.220	0.919	0.925	0.807	0.491	0.819
uk-2002	0.730	0.724	0.674	0.047	0.986	0.985	0.928	0.698	0.970
Friendster	0.640	0.623	0.569	0.361	oom	oom	oom	0.013	0.748
uk-2007-05	0.873	0.877	0.816	0.279	oom	oom	oom	0.132	0.986

RelaxMap is outperformed by DSLM-Map. DSLM-Map finds better clusterings than GossipMap on all graphs except for LiveJournal, on the two largest graphs by a significant margin. On no graph, DSLM-Map has a map equation score more than 0.1 worse than the best algorithm. Since modularity and map equation feature counterintuitive behavior like the resolution limit, quality scores on their own can be misleading. We therefore also compare the obtained clusterings in terms of ARI.

Among all runs of all algorithms that optimize the same quality measure, we determine for each graph the detected clustering with best score. For each graph and quality measure, we use this best clustering as baseline to which we compare all other detected clusterings that were detected optimizing the same quality measure. Table 6.3 shows the average similarity in terms of adjusted rand index and highlights which algorithm detected the used baseline clustering. In most cases, this is the sequential baseline.

For modularity, this is in contrast to the results from Table 6.2 where on average, PLM outperforms Louvain for most graphs. Only on Friendster and uk-2002, the modularity-optimizing algorithm that found the best clustering also has the highest average quality scores. We observe that the modularity-optimizing algorithms do not consistently find the same clustering. Clusterings may vary vastly depending on the random seed. Further, on social networks the adjusted rand indices are smaller than on web graphs. This is probably due to web graphs having a more pronounced community structure. DSLM-Mod produces clusterings that are less similar, but still much more similar than DSLM-Mod w/o Cont., which produces vastly different clusterings. This confirms our observation from Table 6.2. Omitting the contraction significantly decreases the quality of clusterings on real world graphs.

Infomap is in general much more stable than Louvain with an adjusted rand index close to 1. DSLM-Map produces very similar clusterings on uk-2002. On the LiveJournal and Orkut graphs, the clusterings are slightly less similar. Quite interestingly, the parallel RelaxMap and the distributed GossipMap algorithms produce significantly different clusterings in particular for the two social networks. As the results of the directed Infomap algorithm show, this is not due to optimizing the directed map equation. We conclude that RelaxMap and GossipMap indeed fail to find similar clusterings reliably.

Comparison to Recent Approaches. In this section, we provide a short comparison to some approaches that were developed at the same time as or after our algorithms [Ham+18a] based on data published by the respective papers.

For modularity, we compare DSLM-Mod to Vite [Gho+18b; Gho+18a]. They publish results for three graphs we also consider – uk-2007-05, Friendster and Orkut. On similar numbers of processors, their approach is roughly 2-3 times faster than DSLM-Mod. Their modularity scores are better for Friendster and Orkut, but worse for uk-2007-05.

We are aware of three more recent approaches [Fun19; FA19; ZY18] that optimize the map equation. InfoFlow [Fun19] is based on a simple agglomerative clustering scheme were clusters are merged repeatedly. Experiments for InfoFlow were only executed on a single desktop computer, thus no performance comparison is possible. No quality results for any of the graphs we included were published. For the graphs where they report the map equation score, the score for InfoFlow is frequently more than 0.3 and sometimes more than 0.5 higher (i.e., worse) than their sequential implementation. The approach by Zeng and Yu [ZY18] reports no map equation scores as numbers, but plots show that their scores are slightly worse than the sequentially achieved scores. Concerning the running time, the scalability of their approach seems good while the running time is comparable or higher than for our approach. Faysal and Arifuzzaman [FA19] do not report results on any of the graphs we use. They report differences in map equation score compared to InfoMap of 0.03 up to 0.39 depending on the graph while achieving speedups of up to only 3.31 with up to 256 processors. For the Youtube social network, they report execution times above 100 seconds even with 256 processors. We also included this network in preliminary experiments and got running times of about 15 seconds using just two hosts, i.e., at least five times less cores.

These results suggest that Vite [Gho+18b; Gho+18a] and the approach by Zeng and

Yu [ZY18] are the best more recent competitors that should be compared to our approach in future work. As both use a simple partitioning of the nodes based on node ids, an important question is also what influence randomized node ids have on their approach.

6.6 Conclusion

We have introduced two distributed graph clustering algorithms, DSLM-Mod and DSLM-Map, that optimize modularity and map equation, respectively. They are based on the Thrill framework. In an extensive experimental evaluation, we have shown that on LFR benchmark graphs, DSLM-Map achieves excellent results, even better than the sequential Infomap algorithm. For DSLM-Mod, we also evaluate a variant without contraction which has great performance on LFR benchmark graphs. The full DSLM-Mod algorithm with contraction fails to recover the ground truth on LFR benchmark graphs – similar to the sequential Louvain algorithm – but significantly outperforms the variant without contraction on real-world graphs. On real-world graphs, both distributed algorithms find clusterings only slightly different than the sequential algorithms. Compared to GossipMap, the state-of-the-art distributed algorithm for optimizing map equation, DSLM-Map is up to an order of magnitude faster while detecting clusterings that have similar or better map equation scores and are more similar to the clustering with the best map equation score.

In the first local moving phase, synchronous local moving seems to be superior to sequential local moving. Further research is needed to see if this is a phenomenon particular to the LFR graphs we studied or if synchronous local moving could be a way to avoid local maxima when optimizing such quality functions. After the contraction, more careful local moving strategies should be developed though to avoid the problems we see in particular on LFR graphs. Therefore, further research on different local moving strategies seems to be a promising direction.

7 Structure-Preserving Sparsification Methods for Social Networks

This chapter is based on joint work with Gerd Lindner, Henning Meyerhenke, Christian L. Staudt and Dorothea Wagner [Ham+16]. Parts of this chapter have been published in preliminary form in [Lin+15]. Compared to the publication [Ham+16], parts of the results not related to community detection have been removed and we extended the discussion of sparsification with respect to community detection. Further, we added a small proof concerning the running time of our parallel triangle listing algorithm.

Sparsification reduces the size of networks while preserving structural and statistical properties of interest. Various sparsifying algorithms have been proposed in different contexts. In [Lin+15; Ham+16], we contribute the first systematic conceptual and experimental comparison of *edge sparsification* methods on a diverse set of network properties. In this chapter, we focus on properties concerning communities. Our goal is to identify methods that allow us to remove a significant amount of edges while preserving the community structure such that community detection algorithms can still detect it. This would allow us to make community detection algorithms more scalable by running them on a subset of the edges. A requirement for this is of course that the sparsification method itself is scalable, an aspect we also examine in this chapter.

Sparsification methods can be understood as methods for rating edges by importance and then filtering globally or locally by these scores. We show that applying a local filtering technique improves the preservation of all kinds of properties. In addition, we propose a new sparsification method (*Local Degree*) which preserves edges leading to local hub nodes. All methods are evaluated on a set of social networks from Facebook, Google+, Twitter and LiveJournal with respect to network properties including diameter, connected components and community structure. In order to assess the preservation of the community structure, we also include experiments on synthetically generated networks with ground truth communities. Experiments with our implementations of the sparsification methods show that many network properties can be preserved down to about 20% of the original set of edges for sparse graphs with a reasonable density. The experimental results allow us to differentiate the behavior of different methods and show which method is suitable with respect to which property. While our Local Degree method is best for preserving connectivity and short distances, other newly introduced local variants are best for preserving the community structure. While some methods manage to keep intra-community edges in synthetic benchmark graphs, they lead to very different communities on the social networks. There, simple random edge sampling better preserves the community structure.

7.1 Introduction

Most real-world complex networks, including social networks, are already sparse in the sense that for n nodes the edge count m is asymptotically in $\mathcal{O}(n)$. Nonetheless, typical densities lead to a computationally challenging number of edges. Here, we pursue the goal of further sparsifying such networks by retaining just a fraction of edges (sometimes called a "backbone" of the network). In an experimental evaluation, we examine how this affects the community structure of the network as well as properties related to communities like the average clustering coefficient or connected components.

Sparsification can be applied as an acceleration technique: By disregarding a large fraction of edges that are unimportant for the task, running times of network analysis algorithms like community detection algorithms can be reduced. We can also think of sparsification as lossy compression. Large networks can be strongly reduced in size if we are only interested in certain structural aspects that are preserved by the sparsification method.

Other applications include information visualization: Even moderately sized networks turn into "hairballs" when drawn with standard techniques, as the amount of edges is visually overwhelming. In contrast, showing only a fraction of edges can reveal network structures to the human eye if these edges are selected appropriately [NOB15].

From a network science perspective, sparsification can yield valuable insights into the importance of relationships and the participating nodes: Given that a sparsification method tends to preserve a certain property, the method can be used to rank or classify edges, discriminating between essential and redundant edges.

The core idea of the research presented here is that not all edges are equally important with respect to properties of a network: For example, a relatively small fraction of long-range edges typically act as shortcuts and are responsible for the small-world phenomenon in complex networks. The importance of edges can be quantified, leading to *edge scores*, often also referred to as *edge centrality values*. In general, we subsume under these terms any measure that quantifies the importance of an edge depending on its position within the network structure. Sparsification can then be broken down into the stages of (i) edge scoring and (ii) filtering the edges using a global score threshold.

Despite the similar terminology, our work is only weakly related to a line of research in theoretical computer science where *graph sparsification* is understood as the reduction of a dense graph ($\Theta(n^2)$ edges) to a sparse ($\mathcal{O}(n)$ edges) or nearly-sparse graph while provably preserving properties such as spectral properties (e. g. [Bat+13]). The networks of our interest are already sparse in this sense. With the goal of reducing network data size while keeping important properties, our research is related to a body of work that considers sampling from networks (on which [ANK13] provides an extensive overview). Sampling is concerned with the design of algorithms that select edges and/or nodes from a network. Here, node and edge sampling methods must be distinguished: For node sampling, nodes *and* edges from the original network are discarded, while edge sampling preserves all nodes and reduces the number of edges only. The literature on node sampling is extensive, while pure edge sampling and filtering techniques have not been considered as often. A seminal paper [LF06] concludes that node sampling techniques are preferable, but considers few edge sampling techniques. The study presented in [Ebb+08] looks at how well a sample of 5%-20% of the original network preserves certain properties, and is mainly focused on node sampling through graph exploration. It concludes that random walk-based node sampling works best on complex networks, but does so on the basis of experiments on synthetic graphs only and compares only with very simple edge sampling methods.

Only edge sampling techniques are directly comparable to our edge scoring and filtering methods. In this work, we restrict ourselves to reducing the edge set, while keeping all nodes of the original graph. Preserving the nodes allows us to infer properties of each node of the original graph. This is important because in network analysis, the unit of analysis is often the individual node, e. g. when a score for each user in an online social network scenario shall be computed. With respect to the goal of accelerating the analysis, many relevant graph algorithms scale with m so reducing m is more relevant. Also, for community detection, with node sparsification we need a second step to infer the communities of the excluded nodes. In contrast, edge sparsification allows us to directly assign all nodes to communities.

Another related approach is the *Multiscale Backbone* [SBV09], which is applicable on weighted graphs only and is therefore not included in our study. Instead of applying a global edge weight cutoff for edge filtering, which hides important structures at different scales, this approach aims at preserving them at all scales.

A recent[1] study [JS16] with a design similar to ours focuses in detail on algebraic distance for edge rating (see Section 7.3.6).

7.1.1 Contribution

We contribute the first systematic conceptual and experimental comparison of existing and novel edge scoring and filtering methods on a diverse set of network properties. Descriptions and literature references for the related methods which we reimplemented are given in Section 7.2, for some of them we include descriptions of how we parallelized them. In Section 7.2 we also introduce our Local Degree sparsification method and Edge Forest Fire, an adaptation of the existing node sparsification technique to edges. Furthermore, we propose a local filtering step that has been introduced by [SPR11] for one specific sparsification technique as a generally applicable and beneficial post-processing step for preserving the connectivity of the network and most properties we consider.

Our results illuminate which methods are suitable with respect to which properties of a network. We show that our Local Degree method is best for preserving connectivity and short distances which results in a good preservation of the diameter of the network. In [Ham+16], we show that it also preserves some centrality measures and the behavior of epidemic spreading. Depending on the network, our Local Degree method can also preserve clustering coefficients. Concerning the preservation of the community structure, we show that some of the newly introduced variants with local filtering are best for

[1]"Recent" as of the publication of our papers [Lin+15; Ham+16].

preserving the community structure while the variants without local filtering do not preserve the community structure in our experiments.

Furthermore, we have published efficient parallelized implementations and a framework for such methods as part of the NetworKit open-source tool suite [SSM16]. While our study covers various approaches from the literature, it is by no means exhaustive due to the vast amount of potential sparsification techniques. With future methods in mind, we hope to contribute a framework for their implementation and evaluation.

7.2 Edge Sparsification

All edge sparsification methods we consider can be split up into two stages: (i) the calculation of a score for each of the edges in the input graph (where the score is high if the edge is important) and (ii) subsequent filtering according to such an edge score. In this section we define this framework of scoring and filtering edges more formally. Using this framework we introduce Local Filtering as an optionally applicable step. We conclude this section by addressing some limits of edge sparsification.

Formally, we define an edge (centrality) score as follows:

Definition 7.1 (Edge Centrality Scores)**.** *Given a simple, undirected graph $G = (V, E)$, an edge (centrality) score is a function $c_G : E \to \mathcal{A}$ which assigns to each edge $e = \{u, v\} \in E$ an attribute value $x \in \mathcal{A}$ of (at least) ordinal scale of measurement. The assigned value depends on the position of the edge within the network G as defined by a set of edges $E' \subseteq E$. In the following we call such a value an edge score, write $c(e)$ where the graph is implied from the context, and assume that $\mathcal{A} = \mathbb{R}^+$.*

7.2.1 Global and Local Filtering

The simplest way to filter by such an edge score is to apply a global threshold and keep all edges whose score is equal to or above the threshold. For the comparison of the different sparsification methods we need to be able to filter the network such that all methods keep an equal percentage of the edges of the network. As the methods produce scores with different ranges of values and different distributions of these values we cannot simply define a threshold that is the same for all methods. Also using a different threshold per method does not solve the problem as it is possible that many edges have the same score and thus there is no threshold that leads to the desired ratio of kept edges. Therefore, we define global filtering not as applying a given threshold but as filtering the edges such that only a given ratio of the edges remains:

Definition 7.2 (Global Filtering)**.** *Given a graph $G = (V, E)$ and an edge score $c : E \to \mathbb{R}^+$, a global filtering step by ratio $r \in [0, 1]$ reduces the edges in the sparsified graph $G' = (V, E')$ to $\lfloor r \cdot |E| \rfloor$ edges with the highest values of $c(e)$, i. e. $E' \subseteq E$, $|E'| = \lfloor r \cdot |E| \rfloor$ and $c(e') \geq c(e) \, \forall e' \in E' \, \forall e \in E \setminus E'$.*

For an implementation of global filtering, it is enough to sort all edges by the edge score in descending order and keep the first $\lfloor r \cdot |E| \rfloor$ edges. In order to make sure that

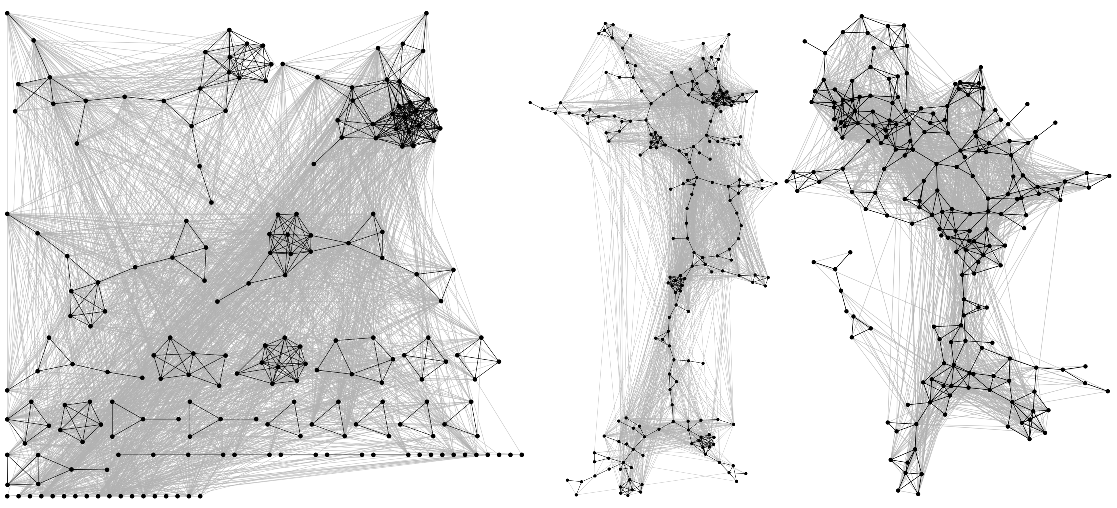

(a) Quadrilateral Simmelian Backbone (b) With UMST (c) With local filtering

Figure 7.1: Drawing of the Jazz musicians collaboration network according to a variant of the Quadrilateral Simmelian Backbone with 15% of the edges (in black)

in the case of edges with equal score a given order in the graph does not influence our results, we sort edges that have an equal score in a random order by using a random edge score as tiebreaker in comparisons.

A problem with global filtering as described above is that methods that are based on local measures like the number of quadrangles an edge is part of tend to assign different scores in different parts of the network as e. g. some parts of the network are much denser than other parts. Sparsification techniques like (Quadrilateral) Simmelian Backbones [NOB14; NOB15] use different kinds of normalizations of quadrangles (see below for details) in order to compensate for such differences. Unfortunately, these normalizations still do not fully compensate for these differences. In Figure 7.1a we visualize the Jazz network [GD03] with 15% kept edges as an example. As one can see in the figure, many nodes are isolated or split into small components, the original structure of the network (shown with gray edges) is not preserved.

Simmelian Backbones have been introduced for visualizing networks that are otherwise hard to layout. For layouts, it is important to keep the connectivity of the network as otherwise nodes cannot be positioned relative to their neighbors. In order to preserve the connectivity, [NOB15] keep the union of all maximum spanning trees (UMST) in addition to the original edges. In Figure 7.1b we show the result when we keep the UMST. While the network is obviously connected, much of the local structure is lost in the areas between the dense parts – which is not surprising as we only added the union of some trees.

[SPR11] face a similar problem as with their sparsification technique based on Jaccard Similarity (see below for details) they want to preserve the community structure. They propose a different solution: Each node u keeps the top $\lfloor \deg(u)^\alpha \rfloor$ edges incident to u, ranked according to their similarity where $\deg(u)$ is the degree of u and $\alpha \in [0, 1]$. An edge is kept when it is ranked high enough by one of the nodes that are incident to it. This procedure ensures that at least one incident edge of each node is retained and thus prevents completely isolated nodes.

In order to define this filtering step formally, we first introduce a formal definition of the rank of node v in the neighborhood of node u:

Definition 7.3 (Neighborhood Rank). *Given a graph $G = (V, E)$ and an edge score $c : E \to \mathbb{R}^+$, the neighborhood-rank $r_c(u, v)$ is the position of v in the sorted list of neighbors of u, i. e. $r_c(u, v) := |\{x \in N(u) : c(\{u, x\}) < c(\{u, v\})\}| + 1$.*

Note that when two edges that are incident to u have the same edge score they also have the same rank. Again, we want to be able to keep an exact ratio of edges. In order to achieve this we transform this local rank into a score that can be used for global filtering.

Definition 7.4 (Local Filtering Score). *Given a graph $G = (V, E)$ and an edge score $c : E \to \mathbb{R}^+$, the directed local filtering score $l_c : V \times V \to [0, 1]$*

$$l_c(u, v) := \begin{cases} 1, & \textit{if } \deg(u) = 1 \\ 1 - \log(r_c(u, v))/\log(\deg(u)), & \textit{otherwise} \end{cases}$$

Then $l_c(\{u, v\}) := \max(l(u, v), l(v, u))$ is the local filtering score that belongs to c.

Lemma 7.5. *Filtering locally according to an edge score c with parameter α is equivalent to keeping all edges with $l_c(\{u, v\}) \geq 1 - \alpha$.*

Proof. An edge $\{u, v\}$ is kept by local filtering iff $r_c(u, v) \leq \deg(u)^\alpha$ or $r_c(v, u) \leq \deg(v)^\alpha$, as the top $\deg(u)^\alpha$ (or $\deg(v)^\alpha$) edges are kept. If $\deg(u) = 1$, the only edge that is incident to u is always kept as $1^\alpha = 1$. For $\deg(u) > 1$, it holds that

$$\begin{aligned} r_c(u, v) &\leq \deg(u)^\alpha \\ \Leftrightarrow \log(r_c(u, v)) &\leq \log(\deg(u)) \cdot \alpha \\ \Leftrightarrow \frac{\log(r_c(u, v))}{\log(\deg(u))} &\leq \alpha \\ \Leftrightarrow \underbrace{1 - \frac{\log(r_c(u, v))}{\log(\deg(u))}}_{= l_c(u,v)} &\geq 1 - \alpha \end{aligned}$$

This shows the claim, as $l_c(\{u, v\})$ is exactly defined as the maximum of $l_c(u, v)$ and $l_c(v, u)$, which means that global filtering by $l_c(\{u, v\})$ will keep the edge iff one of the directions would have been kept by local filtering. □

In Algorithm 7.1 we show the parallel algorithm that we use to transform edge scores for local filtering. We iterate over all nodes in parallel, sort the neighbors, determine the rank for each neighbor and assign its score. As the two scores of an edge are possibly

```
**Input:** Graph $G = (V, E)$, edge score $s : E \to \mathbb{R}$
**Output:** Edge score $l : E \to [0, 1]$
1  $l(u, v) \leftarrow 0 \,\forall \{u, v\} \in E$;
2  foreach $u \in V$ do in parallel
3      $r \leftarrow 0$; // rank
4      $s \leftarrow 1$; // number of equal scores in a row
5      $o \leftarrow -\infty$; // old score
6      foreach $v \in N(u)$ sorted by $s(u, v)$ in descending order do
7          if $s(u, v) \neq o$ then
8              $r \leftarrow r + s$;
9              $s \leftarrow 1$;
10         else
11             $s \leftarrow s + 1$;
12         if $\deg(u) > 1$ then
13             $e \leftarrow 1 - \log(r)/\log(\deg(u))$;
14         else
15             $e \leftarrow 1$;
16         $l(u, v) \leftarrow$ atomic max of $l(u, v)$ and $e$;
```

Algorithm 7.1: Transformation of global edge scores into edge scores that are equivalent to local filtering

calculated at the same time, we use an atomic maximum for assigning the final score. The total time complexity is $\mathcal{O}(m \cdot \log(d_{\max}))$, where $d_{\max}$ denotes the maximum degree in G, as every list of neighbors needs to be sorted.

In Figure 7.1c we show the Jazz network, sparsified again to 15% of the edges with the Quadrilateral Simmelian Backbone method using local filtering. With the local filtering step, the network is almost fully connected and local structures are maintained, too. Additionally, as already Satuluri et al. observed when they applied local filtering to their Jaccard Similarity, the edges are much more evenly distributed among the different parts of the network. This means that we can still see the local structure of the network in many parts of the network and do not only maintain very dense but disconnected parts. An explanation for this is that a node u has at least degree $\deg(u)^{\alpha}$ in the sparsified graphs which means high-degree nodes loose more incident edges than low-degree nodes but high-degree nodes still keep more neighbors than low-degree nodes. Therefore, some properties are kept but still the differences are smoothened in order to ensure that we retain structure in every part of the network. In our evaluation we confirm that many properties of the considered networks are indeed better preserved when the local filtering step is added. Furthermore, we show that the local filtering step leads to an almost perfect preservation of the connected components on all considered networks even though this is not inherent in the method. This suggests that local filtering is superior to preserving a UMST as not only connectivity but also local structures are preserved.

As one of our contributions we therefore propose to apply this local filtering step to all sparsification methods and not only to Jaccard Similarity, where the local filtering step has been introduced. In our evaluation we do not further consider the alternative of preserving a UMST as preliminary experiments have shown that adding a UMST has no significant advantage over the local filtering step in terms of the preservation of network properties. With local filtering, our sparsification pipeline consists of the following stages: (i) calculation of an edge score, (ii) conversion of the edge score into a local edge score and (iii) global filtering. In the evaluation, we prefix the abbreviations of the local variants with “L”.

7.2.2 Limits of Edge Sparsification

While we show in our work that edge sparsification can be successfully applied to many social networks, we also want to mention the limits of these sparsification approaches.

Let us consider a network that has two times as many edges as nodes, i. e. the average degree is four. If we remove 50% of the edges and the network is still connected, we have a tree with one additional edge. This means that almost all triangles, which are characteristic for a social network, are destroyed. Also, a possibly existing community structure that is defined by being particularly dense compared to the rest of the network cannot be maintained in a tree as every community is either a tree or disconnected and it is therefore not possible to discriminate between different connected communities based on density. Therefore, one cannot expect to maintain the properties of the network when the number of edges is equal to (or even less than) the number of nodes.

Some of the sparsification methods we present rely on particular structures that are present in many real-world social networks. For our novel method, local degree, we exploit the presence of meaningful hubs that have a relatively high degree. Other methods assume the presence of triangles or quadrangles that describe the community structure of the network. While it is possible that networks do not have these structures, these structures are fundamental characteristics of social networks. Therefore, in the context of social networks relying on these structures is no limitation.

In terms of the memory usage some of the sparsification methods we present need additional memory, but at maximum a separate copy of the graph including edge ids and edge scores. Some of them also use additional memory per thread in the parallel computation, but then only one bit per node, the neighborhood of one node or a queue of in the worst case all nodes. Therefore, memory usage can be a limitation if the considered graph is almost as large as the available memory – but then also the memory needed for storing the edge scores itself might not be available. It is possible to adapt some of these algorithms to use only a limited amount of internal memory and external memory like a HDD or an SSD instead. For example for triangle counting, a building block of some of the sparsification methods we consider, efficient external memory algorithms exist [HTC14]. Developing sparsification algorithms for external memory or also distributed settings is therefore an interesting topic for future work in order to further push the limits of sparsification in terms of computational resources and scale them to graphs that do not fit into the RAM of a single computer anymore.

Table 7.1: Summary of the methods and the novel local variants we introduce

Method	Novel	Novel local variant
Random Edge	○	●
Jaccard Similarity	○	○
Simmelian Backbones	○	●
Edge Forest Fire	●	●
Algebraic Distance	○	●
Local Degree	●	○

7.3 Sparsification Methods

In this section we describe the sparsification methods that we use in this work. We describe the existing sparsification methods random edge filtering, Jaccard Similarity, Simmelian Backbones and Algebraic Distances. Further we introduce Edge Forest Fire and Local Degree as novel sparsification methods. For all methods where this had not been proposed before we propose to use local filtering as post-processing step. We also present parallel implementations for all methods, though some of them are only partial parallelizations. As most parallelizations are straightforward, we cannot exclude that they have already been used in other implementations. The details are described in the following sections while Table 7.1 gives an overview of all considered methods.

7.3.1 Random Edge (RE)

When studying different sparsification algorithms, the performance of random edge selection is an important baseline. As we shall see, it also performs surprisingly well. The method selects edges uniformly at random from the original set such that the desired sparsification ratio is obtained. This is equivalent to scoring edges with values chosen uniformly at random. Naturally, this needs time linear in the number of edges and calculating the edge scores can be trivially parallelized. Further, if we know the ratio of edges to keep, random edge selection can also be performed in a streaming fashion while reading a graph. Thus, random edge sampling already scales to graphs that do not fit into the RAM of a single computer anymore.

7.3.2 Triangles

Especially in social networks, triangles play an important role because the presence of a triangle indicates a certain quality of the relationship between the three involved nodes. The sociological theory of Simmel [SW50] states that "triads (sets of three actors) are fundamentally different from dyads (sets of two actors) by way of introducing mediating effects." In a friendship network, it is likely for two actors with a high number of common friends to be friends as well. Filtering globally by triangle counts tends to destroy local structures, but several of the following sparsification methods are based on the triangles edge score $T(u, v)$ that denotes for an edge $\{u, v\}$ the number of triangles it belongs to. The time needed for counting the number of all triangles is $\mathcal{O}(m \cdot a)$ [OB14], where a is the graph's arboricity [CN85].

Parallelization. We use a novel parallelized variant of the algorithm introduced by [OB14]. This variant is different from the parallel variant introduced in [ST15] as they need additional overhead in the form of sorting operations or atomic operations for storing local counters which we avoid. Algorithm 7.2 contains the pseudo-code for our algorithm. The algorithm needs a node ordering. While $N(u)$ denotes all neighbors of u, $N^+(u)$ denotes the neighbors of the node that are higher in the ordering. With a smallest-first ordering that is obtained by iteratively removing nodes of minimum degree, $|N^+(u)|$ is bounded by $2a - 1$ [ZN99], which directly gives the running time of $\mathcal{O}(m \cdot a)$. Simply ordering the nodes by degree is actually faster in practice though as noticed by [OB14]. Therefore, we use such a simple degree ordering. For this, we can still show the running time of $\mathcal{O}(m \cdot a)$. Lin, Soulignac and Szwarcfiter show [LSS12, Corollary 5] that

$$\sum_{v \in V} \sum_{w \in N(v)} h(w) \leq 8 \cdot a \cdot m$$

where $h(w)$ is defined as the number of neighbors of w with a degree equal to or higher than that of w. Thus, $|N^+(u)| \leq h(u)$ and the total running time is $\mathcal{O}(m \cdot a)$ as claimed even for a simple degree ordering.

In contrast to [OB14] we count each triangle three times which does not increase the asymptotic running time. In each iteration step of the outer loop we encounter each triangle u is part of exactly once. Therefore, it is enough to count the triangle for the edges that are incident to u and where u has the higher node id. This avoids multiple accesses to the same edge by several threads, we therefore do not need any locks or atomic operations. In the same way we could also update triangle counters per node e. g. for computing clustering coefficients without additional work and without using locks or atomic operations. Note that node markers are thread-local.

```
1 foreach $u \in V$ do in parallel
2     Mark all $v \in N(u)$;
3     foreach $v \in N(u)$ do
4         foreach $w \in N^+(u)$ do
5             if $w$ is marked then
6                 Count triangle $u$, $v$, $w$;
7     Un-mark all $v \in N(u)$;
```

Algorithm 7.2: Parallel triangle counting

7.3.3 (Local) Jaccard Similarity (JS, LJS)

[SPR11] propose a local graph sparsification method with the intention of speedup and quality improvement of community detection. They suggest reducing the edge set to 10-20% of the original graph and use the Jaccard measure to quantify the overlap between node neighborhoods $N(u)$, $N(v)$ (i.e. the sets of nodes adjacent to u or v) and thereby the (Jaccard) similarity of two given nodes:

$$\mathrm{JS}(u, v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} = \frac{T(u, v)}{\deg(u) + \deg(v) - T(u, v)},$$

The time needed for calculating the Jaccard Similarity is the time for counting all triangles. The authors also propose a fast approximation which runs in time $\mathcal{O}(m)$.

For this Jaccard Similarity, [SPR11] propose the local filtering technique that we have already explained above and that we denote by LJS. The time needed for calculating this local edge score is the time for calculating the Jaccard Similarity and for sorting the neighbors of all nodes, which can be done in $\mathcal{O}(m \log(d_{\max}))$.

This sparsification technique has also been adapted for accelerating *collective classification*, i. e. the task of inferring the labels of all nodes in a graph given a subset of labeled nodes [SRD13].

Parallelization. With our parallel triangle counting variant and pre-calculated node degrees, the edge scores for Jaccard Similarity can be calculated in parallel. We are not aware of any prior work that calculated the Jaccard Similarity in parallel. As shown in Algorithm 7.1 also the local filtering can be parallelized. As we will show in our experimental evaluation the achieved running times with our parallel implementation are very good and not far from random edge filtering.

7.3.4 Simmelian Backbones (TS, QLS)

The *Simmelian Backbones* introduced by [Nic+13] aim at discriminating between edges that are placed within dense subgraphs and those between them. The original goal of these methods was to produce readable layouts of networks. To achieve a "local assessment of the level of actor neighborhoods" [Nic+13], the authors propose the following approach, which we adapt to our concept of edge scores. Given an edge scoring method S and a node u, they introduce the notion of a rank-ordered neighborhood as the list of adjacent neighbors sorted by $S(u, \cdot)$ in descending order. The original *(Triadic) Simmelian Backbone* uses triangle counts T for S. The newer *Quadrilateral Simmelian Backbone* by [NOB15] uses *quadrilateral edge embeddedness*, which they define as

$$Q(u, v) = \frac{q(u, v)}{\sqrt{q(u) \cdot q(v)}}$$

with $q(u, v)$ being the number of quadrangles containing edge $\{u, v\}$ and $q(u)$ being the sum of $q(u, v)$ over all neighbors v of u. They argue that this modified version performs even better at discriminating edges within and between dense subgraphs.

On top of the rank-ordered neighborhood graph that is induced by the ranked neighborhoods of all nodes, Nick et al. [Nic+13] introduce two filtering techniques, a parametric one and a non-parametric one. Like [NOB15], we use only the non-parametric variant. By *TS*, we denote the Triadic Simmelian Backbone and by *QLS* the Quadrilateral Simmelian Backbone. The non-parametric variant uses the Jaccard measure similar to Local Similarity but, instead of considering the whole neighborhood, they use the maximum of the Jaccard measure of the top-k neighborhoods for all possible values of k. While the time needed for quadrangle counting is equal to the time for triangle counting [CN85], the overlap and Jaccard measure calculation of prefixes needs time $\mathcal{O}(m \cdot d_{\max})$ as it needs to be separately calculated for all edges.

Parallelization. Our implementation of triangular Simmelian Backbones is fully parallelized, we use our parallel triangle counting implementation, then we sort all neighborhoods in parallel. Using this information we can compute the triangular Simmelian Backbone scores in parallel for all edges. For the quadrilateral Simmelian Backbones, the quadrangle counting step is sequential as we are not aware of an efficient, parallelized quadrangle counting algorithm on edge level but the remaining part of the algorithm is parallelized as in the case of triangular Simmelian Backbones.

7.3.5 Edge Forest Fire (EFF)

The original Forest Fire node sampling algorithm [LF06] is based on the idea that nodes are "burned" during a fire that starts at a random node and may spread to the neighbors of a burning node. Contrary to random walks a fire can spread to more than one neighbor but already burned neighbors cannot be burned again by the same fire. Each fire thus corresponds to a tree. The basic intuition is that nodes and edges that get visited by more of these walks than others are more important. In order to filter edges instead of nodes, we introduce a variant of the algorithm in which we use the frequency of visits of each edge as a proxy for its relevance.

Algorithm 7.3 shows the details of the algorithm we use to compute the edge score. The fire starts at a random node which is added to a queue. The fire always continues at the next extracted node v from the queue and spreads to neighboring unburned nodes until either all neighbors have been burned or a random probability we draw is above a given burning probability threshold p. The number of burned neighbors thus follows a geometric distribution with mean $p/(1-p)$. When the queue is empty, we reset the burned markers and start a new fire at a random node by adding it to the queue.

As the total length of all walks is hard to estimate in advance, we cannot give a tight bound for the running time.

Parallelization. We use a very simple parallelization for our Edge Forest Fire algorithm. We burn several fires in parallel with separate burn markers per thread and atomic updates of the burn frequency. In order to avoid too frequent updates we update the global counter

```
Input: targetBurnRatio $\in \mathbb{R}$, $p \in [0,1)$
1  edgesBurned $\leftarrow 0$;
2  while edgesBurned $< m\cdot$ targetBurnRatio do
3      Add random node to queue;
4      while queue not empty do
5          $v \leftarrow$ node from queue;
6          while true do
7              $q \leftarrow$ random element from$[0,1)$;
8              if $q > p$ or $v$ has no un-burned neighbors then
9                  break;
10             $x \leftarrow$ random un-burned neighbor of $v$;
11             Mark $x$ as burned;
12             Add $x$ to queue;
13             Increase edgesBurned;
14             Increase burn counter of $\{v, x\}$;
15     Reset burned markers;
```

Algorithm 7.3: Edge Forest Fire

for the number of edges burned only after a fire has stopped burning before we start the next fire.

7.3.6 Algebraic Distance (AD)

Algebraic distance [CS11] (α) is a method for quantifying the structural distance of two nodes u and v in graphs. Its essential property is that $\alpha(u,v)$ decreases with the number of paths connecting u and v as well as with decreasing lengths of those paths. Algebraic distance therefore measures the distance of nodes by taking into account more possible paths than e. g. shortest-path distance and with wider in scope than e. g. the Jaccard coefficient of two nodes' immediate neighborhood. Nodes that are connected by many short paths have a low algebraic distance. It follows that nodes within the same dense subgraph of the network are close in terms of α. Algebraic distance can be described in terms of random walks on graphs and, roughly speaking, $\alpha(u,v)$ is low if a random walk starting at u has a high probability of reaching v after few steps. In a straightforward way, algebraic distance can be used to quantify the "range" of edges, with short-range edges (low $\alpha(u,v)$ for an edge $\{u,v\}$) connecting nodes within the same dense subgraph, and long-range edges (high $\alpha(u,v)$ for an edge $\{u,v\}$) forming bridges between separate regions of the graph. Hence, α restricted to the set of connected node pairs is an edge score in our terms, and can be used to filter out long- or short-range edges. We use $1-\alpha(u,v)$ as edge score in order to treat short-range edges as important.

α is computed by performing iterative local updates on d-dimensional "coordinates" of a node. The coordinates are initialized with random values. Then, in each iteration, the coordinates are set to some weighted average of the old coordinates and the average of

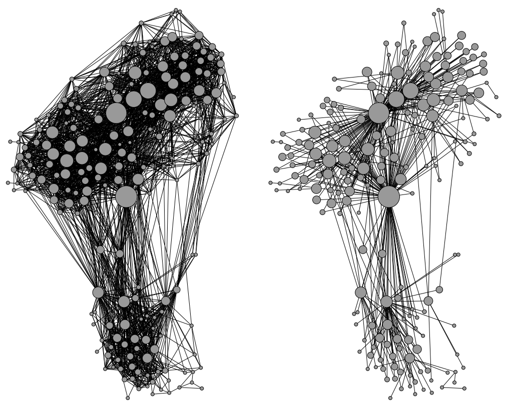


Figure 7.2: Drawing of the Jazz musicians collaboration network and the *Local Degree* sparsified version containing 15% of edges. Node size proportional to degree.

the old coordinates of all neighbors. The algebraic distance is then any distance between the two coordinate vectors, we choose the ℓ_2-norm. As described in [CS11], d can be set to a small constant (e. g. 10) and the distances stabilize after tens of iterations of $\mathcal{O}(m)$ running time each. We choose 20 systems and 20 iterations.

Parallelization. Updates of node coordinates can be easily executed in parallel for all nodes as the new values only depend on the values of the previous round.

7.3.7 Local Degree (LD)

Inspired by the notion of hub nodes, i. e. nodes with locally relatively high degree, and that of local sparsification, we propose the following new sparsification method: For each node $v \in V$, we include the edges to the top $\lfloor \deg(v)^\alpha \rfloor$ neighbors, sorted by degree in descending order. Similar to the local filtering step we explained above we use again $1 - \alpha$ for the minimum parameter α such that an edge is still contained in the sparsified graph as edge score. The goal of this approach is to keep those edges in the sparsified graph that lead to nodes with high degree, i.e. the hubs that are crucial for a complex network's topology. The edges left after filtering form what can be considered a "hub backbone" of the network. In Figure 7.2 we visualize the Jazz network [GD03] as an example.

As only the neighbors of each node need to be sorted, this can be done in $\mathcal{O}(m \log(d_{\max}))$. Using linear-time sorting it is even possible in $\mathcal{O}(m)$ time. We have decided against the linear-time variant and instead parallelized the calculation:

Parallelization. The parallelization of the Local Degree score calculation is very similar to the parallelization for local filtering. We can handle all nodes in parallel. For each node we can independently sort the neighbors and assign the appropriate scores. Again we need to use atomic maximum calculation in order to make sure that parallel updates of edge scores are handled correctly.

7.4 Implementation

For this study, we have created efficient C++ implementations of all considered sparsification methods, and have accelerated them using OpenMP parallelization. All methods, with exception of the inherently sequential quadrangle counting algorithms [CN85], have been parallelized. We have implemented the algorithms in *NetworKit* [SSM16], they are available as part of NetworKit at `https://networkit.github.io`.

Gephi [BHJ09] is a graph visualization tool which we use not only for visualization purposes but also for interactive exploration of sparsified graphs. To achieve said interactivity, we implemented a client for the *Gephi Streaming Plugin* in NetworKit. It is designed to stream graph objects from and to Gephi utilizing the JSON format. Using our implementation in NetworKit, a few lines of Python code suffice to sparsify a graph, calculate various network properties, and export it to Gephi for drawing. The approach of separating sparsification into edge score calculation and filtering allows for a high level of interactivity by exporting edge scores from NetworKit to Gephi and dynamic filtering within Gephi.

For the drawings of the Simmelian Backbones we use *visone*[2].

7.5 Experimental Study

Our experimental study consists of two parts. In the first part (Section 7.5.2) we compare correlations between the calculated edge scores on a set of networks. In the second part (Section 7.5.3) we compare how similar the sparsified networks are to the original network by comparing certain properties of the networks.

7.5.1 Setup

Our experiments have been performed on a compute server with 4 physical Intel Core i7 cores at 3.4 GHz, 8 threads, and 32 GB of memory. For this explorative study, we use the collection of 100 Facebook social friendship networks [TMP12] introduced in Section 2.2.3. They have between 10k and 1.6 million edges, the number of nodes and edges for each network is shown in Figure 7.3. Unless otherwise noted, we aggregate experimental results over this set of networks. The common origin and the high structural similarity among the networks allows us to get meaningful aggregated values.

For the experiments on the preservation of properties we also use the Twitter and Google+ networks [ML12] and the LiveJournal (com-lj) network from [YL12b]. All of them are friendship networks, the Twitter and Google+ networks consist of the combined ego networks of 973 and 132 users, respectively. In Table 7.2 we provide the number of nodes and edges as well as diameter and clustering coefficient averaged over all Facebook networks and the individual values for the three other networks. Furthermore, we provide the number of edges divided by the number of nodes, which indicates how much redundancy there is in the network. As we already mentioned in Section 7.2.2, it is not realistic to

[2]`http://visone.info`

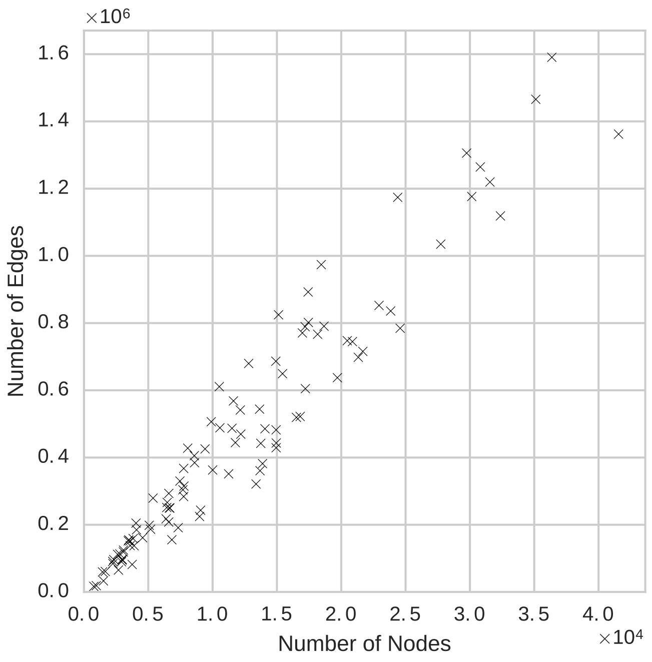


Figure 7.3: Number of nodes and edges of the Facebook networks

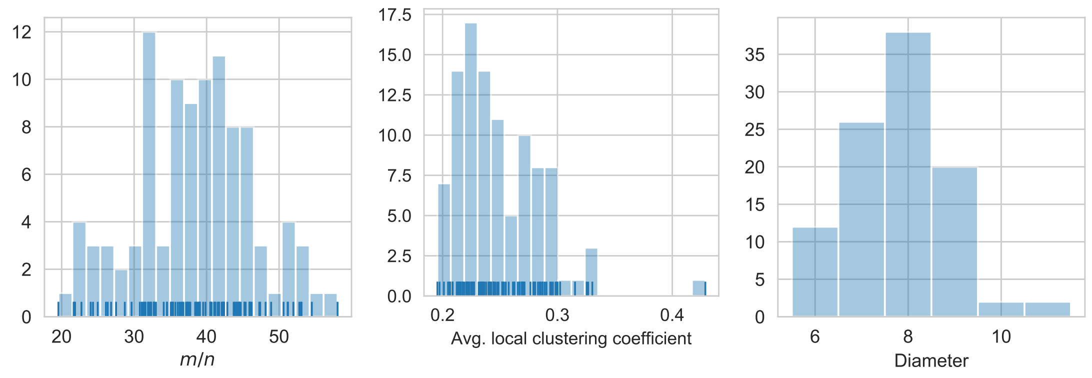
expect that we can preserve the structure of the network (apart from the connectivity) if too few edges remain. The networks we selected have a varying degree of redundancy, but all of them are dense enough such that even if we remove 75% of the edges more than twice as many edges as nodes remain. The characteristics of the Facebook networks are relatively similar. We plot the distributions of m/n, the clustering coefficient, and the diameter in Figure 7.4a, 7.4b and 7.4c.

For the evaluation of the preservation of the community structure we also use networks with known ground truth communities. For this purpose we use the synthetic LFR benchmark [LFR08], see Section 2.1.1 for an introduction.

It remains an open question to what extent results can be translated to other types of complex networks, since according to experience the performance of network analysis algorithms depends strongly on the network structure.

7.5.2 Correlations between Edge Scores

Among our sparsification methods, some are more similar to others in the sense that they tend to preserve similar edges. Such similarities can be clarified by studying correlations between edge scores. We calculate edge score correlations for the set of 100 Facebook networks as follows: For each single network, edge scores are calculated with the various scoring methods and Spearman's rank correlation coefficient is applied. The coefficient is then averaged over all networks and plotted in the correlation matrix (Figure 7.5). There is one column for each method, and the column Mod represents edge scores that are 1



(a) Edge ratio (b) Average local clustering coefficient (c) Diameter

Figure 7.4: Distribution of properties of the Facebook networks. The vertical ticks at the bottom indicate the individual data points except for the diameter, where a bar exists for all unique values.

Table 7.2: Number of nodes n, the number of edges m, m/n, the diameter D and the average local clustering coefficient Cc of the used social networks (average values for the Facebook networks)

Network	n	m	m/n	D	Cc
Facebook	12 083	469 845	38.4	7.8	0.25
com-lj	3 997 962	34 681 189	8.7	21	0.35
gplus	107 614	12 238 285	113.7	6	0.52
twitter	81 306	1 342 296	16.5	7	0.60

for intra-community edges and 0 for inter-community edges after running a modularity-maximizing Louvain community detection algorithm [Blo+08], see Section 1.4.1 for an introduction to the Louvain algorithm. Positive correlations with these scores indicate that the respective rating method assigns high scores to edges within modularity-based communities. The column Tri simply represents the number of triangles an edge is part of. As some of the methods are normalizations of this score, this shows how similar the ranking still is to the original score.

Interpretation of the results is challenging: The correlations we observe reflect intrinsic, mathematical similarities of the rating algorithms on the one hand, but on the other hand they are also caused by the structure of this specific set of social networks (e.g., it may be a characteristic of a given network that edges leading to high-degree nodes are also embedded in many triangles). Nonetheless, we note the following observations: There are several groups of methods. Simmelian Backbones, Jaccard Similarity and Triangles are highly positively correlated which is not unexpected as they are all based on

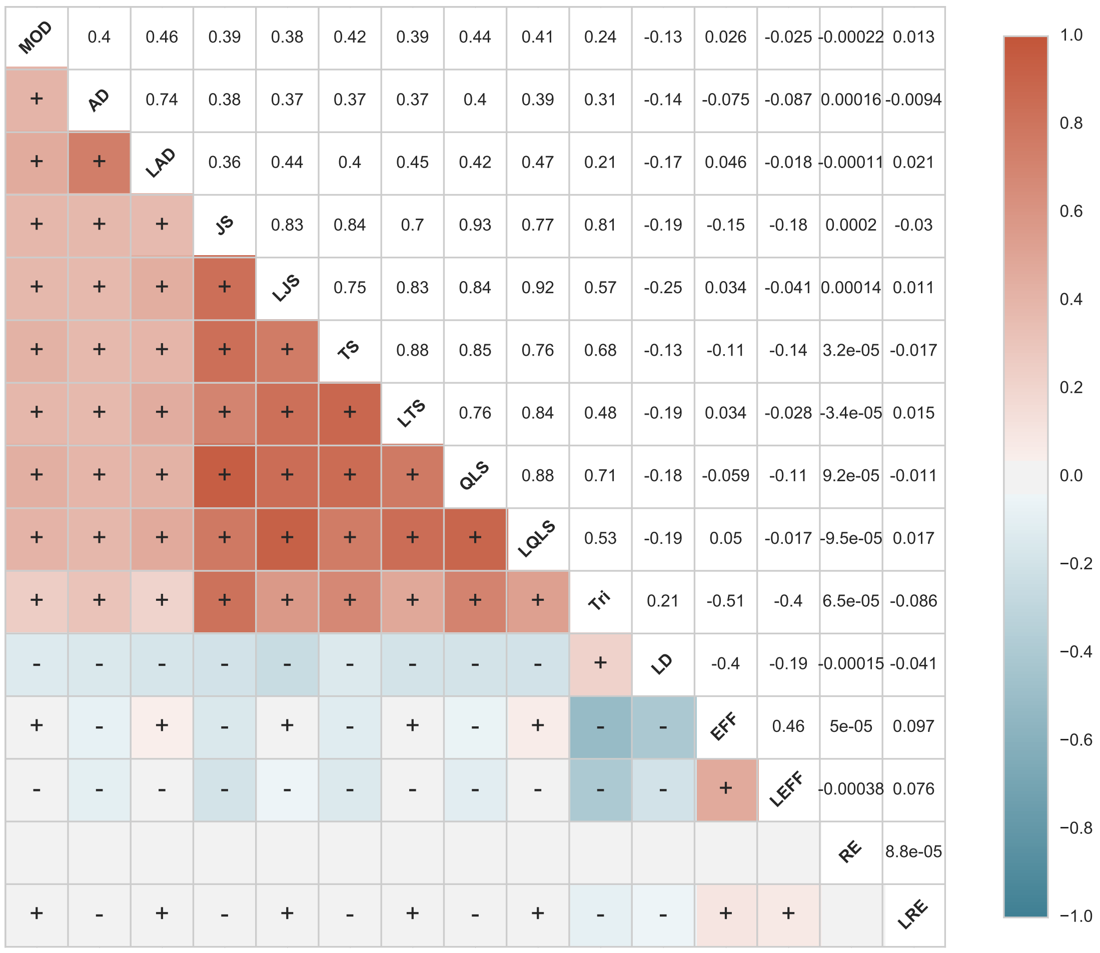


Figure 7.5: Edge score correlations (Spearman's ρ, average over 100 Facebook networks)

triangles or quadrangles and are intended to preserve dense subgraphs. Algebraic distance is still positively correlated with these methods but not as strongly even though they are also intended to prefer dense subgraphs. An explanation for this weaker correlation is that while both prefer dense regions, the order of the individual edges is different. All of the previously mentioned methods are also correlated with the modularity value, algebraic distance with local filtering has the highest score among all of these methods. Our experiments on the preservation of community structure (Section 7.5.3) confirm this relationship. The correlation of the modularity value and these methods are similar to the correlation between algebraic distance and the rest of the methods which shows again that the lower correlation values are probably due to different orderings of the individual edges.

Our new method Local Degree is slightly negatively correlated with all these methods but still positively correlated with the Triangles. It is also slightly negatively correlated with the modularity value, this is due to the method's preference of inter-cluster edges which is also confirmed by our experiments below. The newly introduced Edge Forest Fire is also negatively correlated with Local Degree and even more negatively with Triangles. As expected, random edge filtering is not correlated with any other method.

It is interesting to see that each method is also relatively strongly correlated with its local variant, apart from random edge filtering (we use different random values as basis of the local filtering process). Even the Edge Forest Fire method, which should also be relatively random, has a positive correlation with its local variant. This shows that it prefers a certain kind of edge and that this preference is kept when applying the local filtering.

Among the variants of Simmelian Backbones and Jaccard Similarity also the local variants are more correlated to other local variants than to other non-local variants and also not as strongly correlated to triangles. This shows that the local filtering indeed adds another level of normalization. Also, Jaccard Similarity seems to be more correlated to Quadrilateral Simmelian Backbones than to the variant based on triangles even though Jaccard Similarity is based on triangles itself. This is also interesting to see, as Quadrilateral Simmelian Backbones are computationally more expensive than the Jaccard Similarity.

7.5.3 Similarity in Network Properties

Quantifying the similarity between a network and its sparsified version is an intricate problem. Ideally, a similarity measure should meet the following requirements:

1. *Ignoring trivial differences*: Consider, for example, the degree distribution: One cannot expect the distribution to remain identical after edges get removed during sparsification. It is clear, however, that the general shape of the distribution should remain "similar" and that high-degree nodes should remain high-degree nodes in order to consider the degrees as preserved.

2. *Intuitive and Normalized*: Similarity values from a closed domain like $[0, 1]$ allow for aggregation and comparability. A similarity value of 1 indicates that the property

under consideration is fully preserved, whereas a value of 0 indicates that similarity is entirely lost. In some cases we also used relative changes in the interval $[-1, 1]$ where 0 means unchanged as they provide a more detailed view at the changes.

3. *Revealing Method Behavior*: A good similarity measure will clearly expose different behavior between sparsification methods.

4. *Efficiently computable.*

Following these requirements, we select measures that quantify relative changes for global properties like diameter, size of the largest connected component and quality of a community structure. Further, we analyse the deviation of the average local clustering coefficient from the original value. In our papers [Lin+15; Ham+16], we further consider the ranking of nodes with respect to node centrality scores. As it is not obvious how differences in the ranking of nodes should affect the community structure, we exclude them here.

In the following plots, the measures are shown on the y-axis for a given ratio of kept edges (m'/m) on the x-axis (e.g., a ratio of 0.2 means that 20% of edges are still present). For each value there are two rows of plots. The first contains averages over the 100 Facebook networks with error bars that indicate the standard deviation. The second row contains the values at 20%, 50% and 80% remaining edges of the three other networks. In each row, we show two plots: the left plot with the non-local methods and the right plot with the methods that use local filtering.

Connected Components. All nodes in a *connected component* are reachable from each other. A typical pattern in real-world complex networks is the emergence of a giant connected component which contains most of the nodes and is usually accompanied by some small components. As all of our networks have such a giant component that comprises most nodes, we track its change by dividing the size of the largest component in the sparsified network by the size of the largest component in the original network. As shown in Figure 7.6, out of the non-local methods Edge Forest Fire best preserves the connected components. Random edge deletion leads to a slow decrease in the size of the largest component while Simmelian Backbones, Jaccard Similarity and algebraic distance lead to a separation very quickly. Below 20% of retained edges, the size of the largest component on the Facebook networks drops very quickly, here the networks seem to be decomposed into multiple smaller parts. On the other networks, this drop occurs at different ratios of kept edges which reflects their different densities and probably also their different structures. Local Filtering is able to maintain the connectivity. On the Facebook networks, all methods keep the largest component almost fully connected up to 20% of retained edges, only below that small differences are visible. The results on the LiveJournal, Twitter and Google+ networks show that – as expected – with increasing density it is easier to preserve the connectivity of the network. Our Local Degree method best preserves the connected components of all networks, closely followed by the local variant of random edge deletion and Edge Forest Fire.

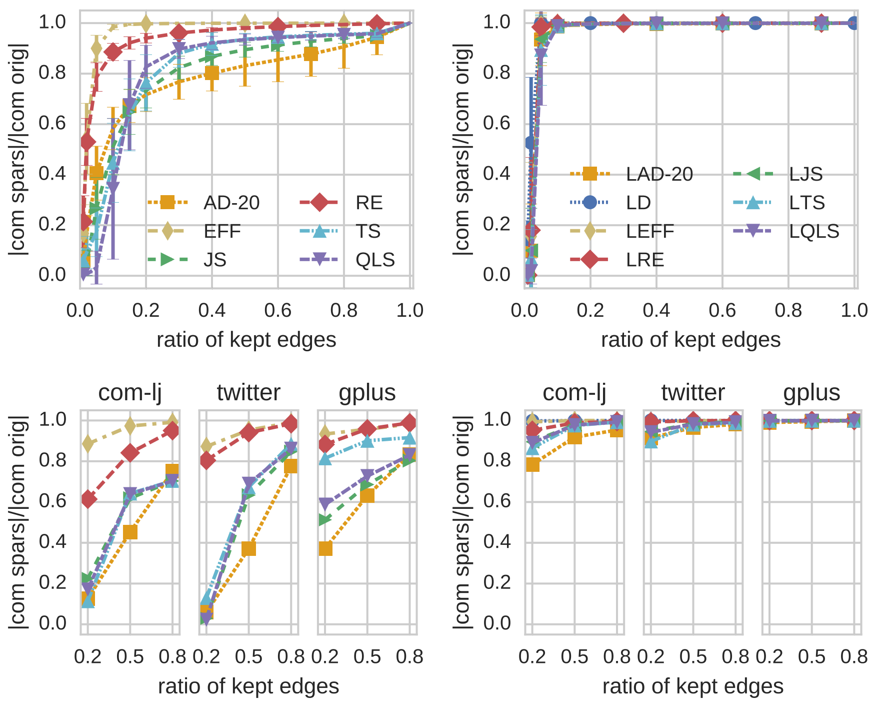


Figure 7.6: The size of the largest component in the sparsified network divided by the size of the largest component in the original network.

For community detection, splitting the graph into smaller connected components might or might not be good. On the one hand, it could be easier to detect communities if they are clearly separated, i.e., if different communities are also different connected components. On the other hand, if a community is separated into several connected components, it cannot be detected as a whole anymore. Examples as shown in Figure 7.1a suggest that we get way too many connected components and thus most likely they split communities. In the following experiments, we examine this in more detail.

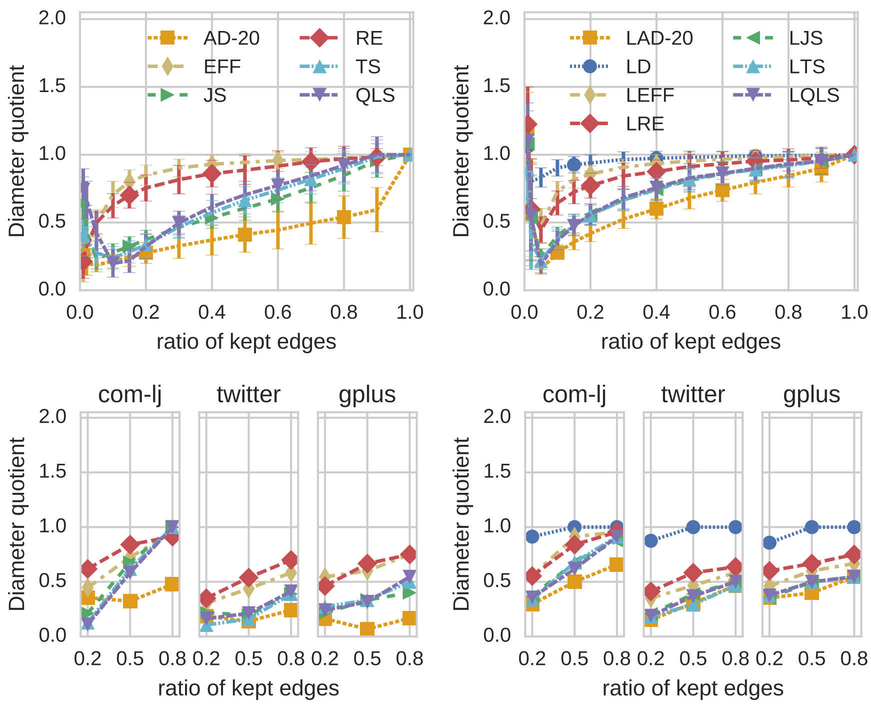
Diameter. The *diameter* of a graph is the maximum length of a shortest path between any two nodes [New10]. The diameter of social networks is often surprisingly small, this is also known as the *small world phenomenon* [WS98]. In case of disconnected graphs, we consider the largest diameter of all connected components. In order to observe how the network diameter changes through sparsification, we plot the quotient of the original network diameter and the resulting diameter, which yields legible results since in practice the diameter is mostly increased during sparsification. We compute the exact diameters using a variation of the ExactSumSweep algorithm [Bor+15] that we contributed to NetworKit.

We motivate the Local Degree method with the idea that shortest paths commonly run through hub nodes in social networks. Therefore, preserving edges leading to high-degree nodes should preserve the small diameter. This is confirmed by our experiments (Figure 7.7a). In contrast, methods that prefer edges within dense regions clearly do not preserve the diameter. With Simmelian Backbones the diameter drops when only few edges are left; this can be explained by the fact that Simmelian Backbones do not maintain the connectivity and that at the end the graph is decomposed into multiple connected components which have a smaller diameter. Algebraic distance is even more extreme in this aspect. Local filtering leads to a slightly better preservation of the diameter when applied to the other methods but algebraic distance remains the worst method in this regard. Note that the LiveJournal network has a higher diameter than the other networks (see Table 7.2); this might explain why the diameter is better preserved there.

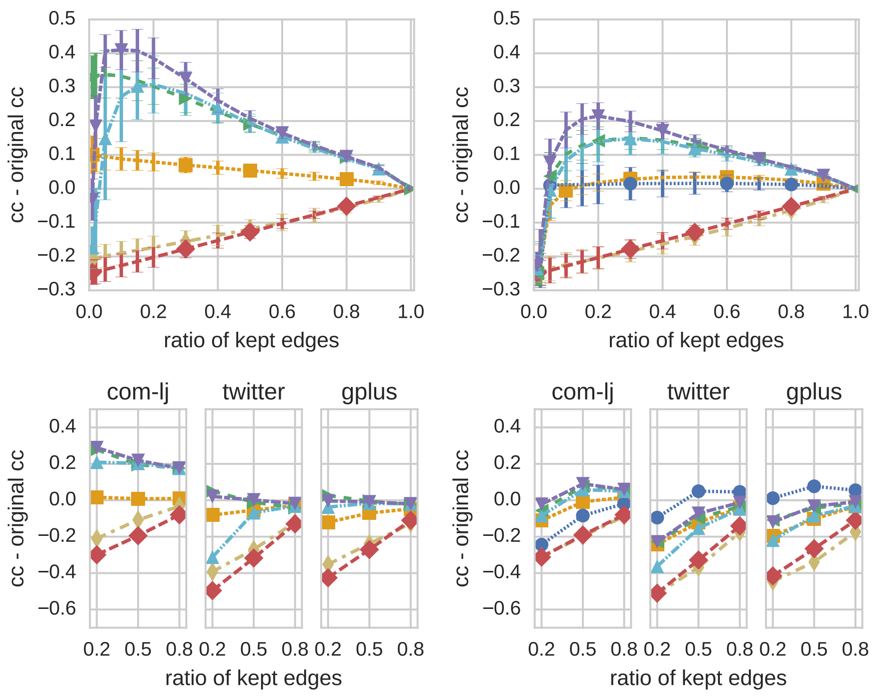
With respect to communities, not preserving the diameter might actually be an advantage. In particular edges connecting otherwise disconnected regions to hub nodes are likely edges between communities. Thus, it is good to see that local filtering does not preserve too many of these edges.

Clustering Coefficient. *Clustering coefficients* are key figures for the amount of transitivity in networks. The *local clustering coefficient* expresses how many of the possible connections between neighbors of a node exist [New10], see Section 1.3.1 for an introduction. Figure 7.7b shows the deviation of the average local clustering from the value of the original network. Both for local and non-local methods we observe three classes of methods on the Facebook networks: methods that clearly decrease the clustering coefficient, methods that preserve the clustering coefficient and methods that increase it.

For both Random Edge and Edge Forest Fire, which are based on randomness, the clustering coefficient drops almost linearly with decreasing sparsification ratio. This can



(a) Original network diameter divided by network diameter



(b) Deviation from original clustering coefficient

Figure 7.7: Preservation of global network properties

also be observed on the other three networks. The additional local filtering step does not significantly change this.

Simmelian Backbones and Jaccard Similarity keep mostly edges within dense regions, which results in increasing clustering coefficients on all networks. Triadic Simmelian Backbones show the weakest increase, on the Twitter network even a decrease of the clustering coefficients. Note that with 0.52 and 0.6 the clustering coefficients are already relatively high on the Google+ and Twitter networks, therefore the very small increase is not surprising. Local filtering slightly weakens this effect on the Facebook networks, on the other networks it is even reversed. Given the high clustering coefficients in the original networks, this is not very surprising as we would need to retain very dense areas while local filtering leads to a more balanced distribution of the edges.

From the previous results especially concerning the connected components one would expect that algebraic distance also increases the clustering coefficients. Interestingly though, filtering using algebraic distance leads to a slight increase of the clustering coefficient on the Facebook networks, constant clustering coefficients on the LiveJournal network and even slightly decreasing clustering coefficients on the Twitter and Google+ networks. With the additional local filtering step, algebraic distance almost preserves the clustering coefficients on the Facebook networks while on the other networks it is slightly decreased. Algebraic distance leads to random noise on the individual edge weights, therefore they probably lead to a more random selection of edges that also destroys more triangles than the selection of Simmelian Backbones and Jaccard Similarity. Our Local Degree method best preserves the clustering coefficient on the Facebook networks, though with some differences between the various networks in the dataset (note the error bars). On the LiveJournal network it leads to a decrease of the clustering coefficient while on the Twitter and Google+ networks it leads to a slight increase of the clustering coefficient. This is probably due to the special structure of ego networks.

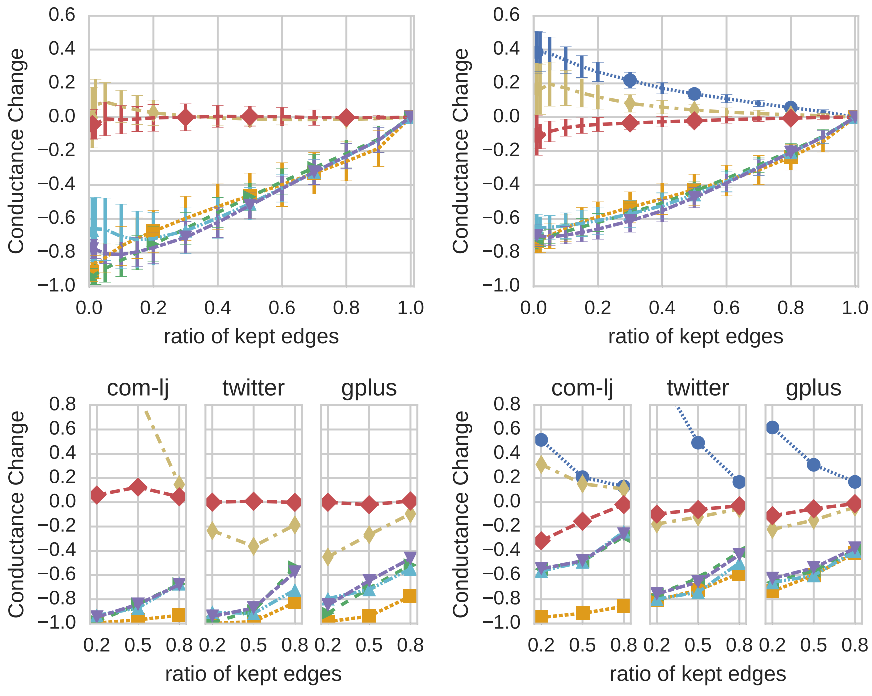
Increasing or constant local clustering coefficients indicate that dense local areas are preserved. While this should be beneficial for community detection, it is not clear if this might not focus too much on very local structures and ignore the broader scope of a community.

Community Structure. For community detection, we use the implementation of the Louvain method with refinement that is part of NetworKit [SM16], see Section 1.4.1 for an introduction. It detects communities of reasonable quality while being fast enough for the vast amount of networks that we get due to the different sparsification methods and ratios of kept edges. In order to understand how the community structure of the networks is maintained, we consider for each network a fixed community structure that has been found by the Louvain method on the original network. We report two crucial properties of this community structure for each level of sparsification. The first is the *conductance*, as introduced in Section 1.3.2, it is a formalization of the fuzzy concept of internally dense and externally sparse communities. Low conductance values indicate clearly separable communities. We consider the average conductance of all communities. Second, we expect that communities are connected. In order to measure this, we introduce the fraction of the nodes in a community that does not belong to the largest connected component of the community as partition fragmentation. We report the average fragmentation of all communities. Intuitively, this score tells us which fraction of a community on average cannot be detected on the sparsified graph due to not being connected to the remaining part.

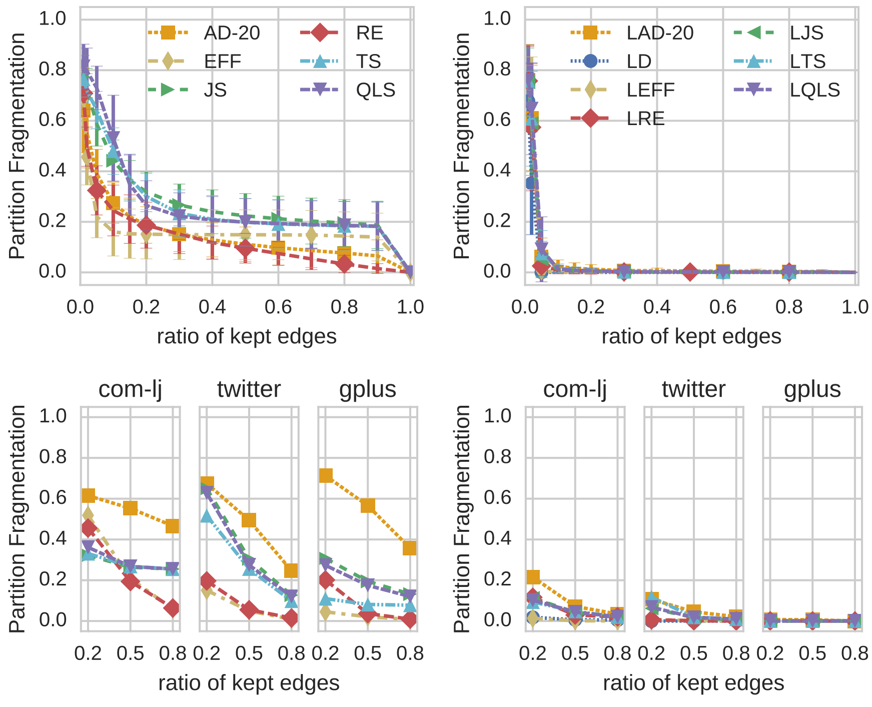
We plot the relative inter-cluster conductance change in Figure 7.8a. A value of 0 means that the conductance stays the same, a value of -1 indicates that the conductance became 0 (i.e. a decrease by 100%) and a value of 1 indicates that the conductance has been doubled (i.e. an increase of 100%). We can again see that there are three categories of algorithms: the first group consisting of random edge sampling preserves the conductance values on most networks. The second group contains only Local Degree and increases the conductance. Edge Forest Fire has no clear behavior. On the LiveJournal network it increases the conductance, while on Twitter and Google+ it rather decreases it. On the Facebook networks, Edge Forest Fire without local filtering preserves the conductance values while with local filtering the conductance values are slightly increased. The third group consisting of Jaccard Similarity, Simmelian Backbones and algebraic distance strongly decreases the conductance. With the additional local filtering step the decrease in conductance is not as strong but still very significant. The keeping of inter-community edges of the Local Degree method, which also explains why it preserves the connectivity so well, can be explained as follows: Consider a hub node x within a community with neighbors that are for the most part also connected to a hub node y with higher degree than x. Due to the way Local Degree scores edges, x will lose many of its connections within the community and may be pulled into the community of a neighboring high-degree node z that is not part of the original community of x. Jaccard Similarity, Simmelian Backbones and algebraic distance on the other hand focus – by design – on intra-community edges. Random edge sampling and Edge Forest Fire filter both types of edges almost equally distributed which is not surprising given their random nature. Depending on the network Edge Forest Fire shows different behavior, this indicates that these networks have a different structure.

In Figure 7.8b it becomes obvious that only local filtering allows methods to keep the intra-cluster connectivity up to very sparse graphs. On the Facebook networks, Simmelian Backbones and Jaccard Similarity without local filtering are actually the worst in this respect, they do not keep the connectivity even though they prefer intra-cluster edges as we have seen before. On the other networks, algebraic distance is even more extreme in this regard. Random edge sampling and Edge Forest Fire on the non-Facebook networks are the only non-local method where a slow increase of the fragmentation can be observed, all other methods lead to a steep increase of the fragmentation during the first 10% of edges that are removed.

Given these observations, we expect that we should be able to find a community structure on the sparsified graph that is very similar to our reference community structure at least if we use Simmelian Backbones, Jaccard Similarity or algebraic distance with local filtering. In Figure 7.9a we compare the community structure that is found by the Louvain method on the sparsified network to the one found on the original network. For

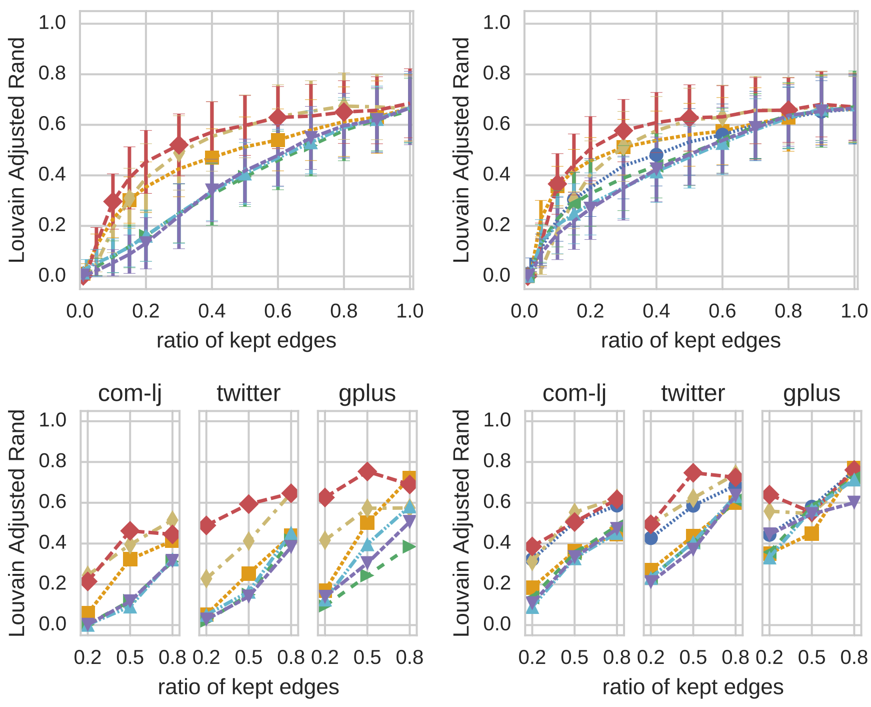


(a) Relative conductance change of a fixed partition



(b) Average partition fragmentation

Figure 7.8: Preservation of community structure properties



(a) Adjusted rand measure between partition into communities on the original network and on the sparsified network

Figure 7.9: Preservation of community structure

this comparison we use the adjusted rand index [HA85] as also introduced in Section 2.3.2. Note that the frequently used normalized mutual information (NMI) measure reports higher similarity values when more communities are detected (see e. g. [VEB09]). This makes it unsuitable for comparing partitions on sparsified networks as we have to expect many small communities when a lot of edges are removed. As [VEB09] also show in their experiments, the adjusted rand index does not have this problem as it has an expected value of 0 for random partitions.

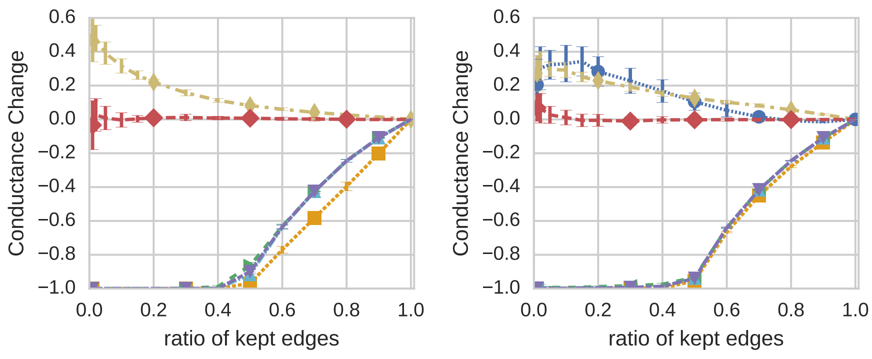
As a first observation we need to note that even when all edges are still in the network (at the right boundary of the plot for the Facebook networks), the community structure found is already different. The Louvain method is randomized, therefore it is not unlikely that every found community structure is different, as we already observed in the experiments for our distributed community detection algorithms in Section 6.5.3. The amount of difference between the community structures even without filtering edges indicates that there is not a single, well-defined community structure in these graphs but many different ones. Preliminary tests with the (slightly slower) Infomap community detection algorithm [RAB09] which has an excellent performance on synthetic benchmark graphs for community detection [LF09b] showed a very similar variance which indicates that this is not due to a weakness of the Louvain algorithm. Filtering the edges such that we can measure that the conductance of one of these many community structures is decreased most probably does not just make this structure clearer but does also lead the algorithm into finding different community structures. Therefore, most sparsification methods lead to significantly different community structures. It is possible that some of the sparsification methods, especially local variants of the Simmelian Backbones, Jaccard Similarity and algebraic distance, simply reveal a different community structure. On the contrary, if all edges are kept with the same probability, the almost same set of community structures can still be found up to a certain ratio of kept edges. Removing less than 40% of the edges at random using random edge sampling or Edge Forest Fire does not seem to lead to significantly more differences than what we observe due to the randomness of the Louvain method. Note that on the three other networks these results are hard to interpret as they are from just a single run, but the general tendencies are similar. Local methods keep the connectivity and are thus slightly better at preserving the community structure.

In order to verify the hypothesis that some differences are due to different community structures being found, we use synthetic networks with ground truth communities that are unambiguous. For this purpose we use the popular LFR generator [LFR08], see Section 2.1.1 for an introduction. As parameters, we choose the configuration with 1000 nodes and small communities from [LF09b]. This means our synthetic networks have a power-law degree distribution with exponent -2, average degree 20 and maximum degree 50. The communities have between 10 and 50 nodes, the community sizes also follow a power-law distribution but with exponent -1. As mixing parameter μ we choose 0.5. This means that each node has as many neighbors in its own community as in all other communities together. For smaller mixing parameters the differences between the different techniques are less obvious, for larger mixing parameters we reach the limits where community detection algorithms are no longer able to identify the ground truth communities. For the plots in Figure 7.10 we use ten different random networks with the same configuration and report again the average and the standard deviation. As we have known ground truth communities for these networks, we use these ground truth communities instead of a community structure found by the Louvain algorithm for the following comparisons.

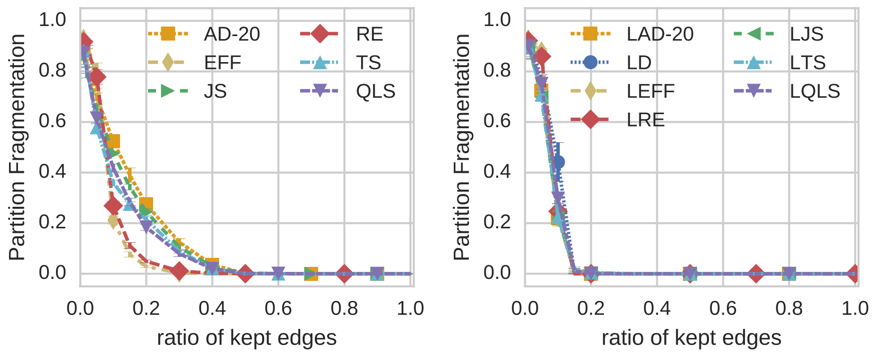
For the inter-cluster conductance in Figure 7.10a the results are similar to the results for the Facebook networks but much clearer. Random edge filtering almost perfectly preserves the inter-cluster conductance both in the variant without and the one with local filtering. Edge Forest Fire and Local Degree lead to a clear increase of the conductance, again independent of the local filtering step. Compared to the Facebook networks this increase for Edge Forest Fire is now much stronger and earlier in the sparsification process. Simmelian Backbones, Jaccard Similarity and algebraic distance lead to a strong decrease of the inter-cluster conductance. With 40% remaining edges, the conductance reaches almost 0 for all of them. If a measure was able to perfectly distinguish between intra- and inter-cluster edges, the inter-cluster conductance could reach 0 when the ratio of kept edges reaches 50%. All methods are not far from that goal, but algebraic distance is the best method in this regard. With a local filtering post-processing step, algebraic distance is more similar to the other methods. For the other methods, only minor changes can be observed. It is visible, though, that some inter-cluster edges seem to remain.

On the LFR networks the connectivity in the communities seems to be preserved much better than on the Facebook networks, see Figure 7.10b. Up to 50% of removed edges, none of the methods leads to any noticeable fragmentation. Only when more edges are removed, the Simmelian Backbones, Jaccard Similarity and algebraic distance seem to disconnect parts of the communities. With the additional local filtering step the connectivity inside communities is almost perfectly preserved up to 15% remaining edges. As the networks have an average degree of 20 we also cannot expect that connectivity is preserved much further as with 10% remaining edges only a tree could be preserved.

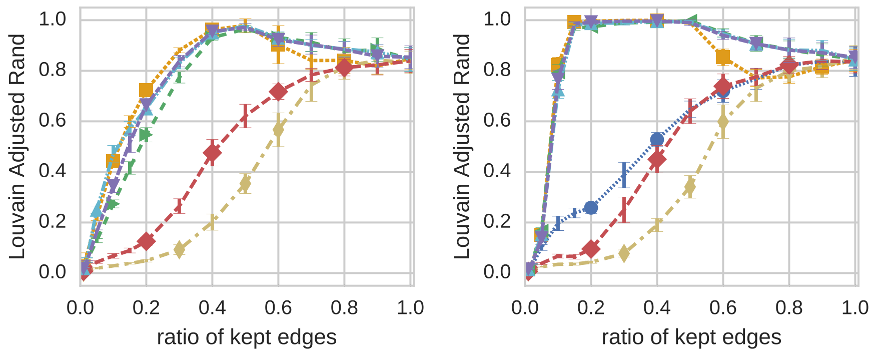
In Figure 7.10c we compare the ground truth communities to the community structure found by the Louvain algorithm (again with refinement). Without the sparsification step, the Louvain algorithm is unable to detect the ground truth communities, this is most likely due to the resolution limit of modularity [FB07]. On the generated networks with clear ground truth communities, our intuition that random edge, local degree and edge forest fire should not be able to preserve the community structure is verified. While removing less than 20% of the edges leads to similar detection rates, the differences increase as more and more edges are removed. Edge Forest Fire is worst at keeping the community structure on LFR networks. On the contrary those methods that show positive results for the preservation of the community structure actually lead to sparsified networks where the Louvain algorithm is better able to find the community structure. As we could expect from the partition fragmentation, without local filtering the adjusted rand index decreases after 50% of the edges have been removed. With local filtering, though, there is a range between 50% and 15% of remaining edges where the Louvain algorithm can almost exactly recover the ground truth communities on the sparsified



(a) Relative conductance change of the ground truth communities



(b) Average partition fragmentation of the ground truth communities



(c) Adjusted rand measure between ground truth communities and found communities on the sparsified network

Figure 7.10: Preservation of the community structure on the generated LFR graphs

network. This shows that sparsification can even increase the quality of communities found by community detection algorithms. Algebraic distance with local filtering seems to work best in this regard. During the first 50% of removed edges algebraic distance shows a strange behavior, though – especially with local filtering the detection rate first drops a bit. This is surprising as algebraic distance leads to the strongest decrease of the inter-cluster conductance right at the beginning. A possible explanation is that the Louvain algorithm merges especially small clusters. If other methods filter edges between these small clusters first, this most probably helps the Louvain algorithm most.

With all these experiments we have seen that measuring the preservation of the community structure is a challenging task especially when no ground truth communities are known. Our results suggests that the social networks either do not contain a clear community structure or that the Louvain algorithm is unable to identify this structure. Random edge deletion and edge forest fire seem to preserve this uncertainty in the sense that the Louvain algorithm still identifies relatively similar communities. Simmelian Backbones, Jaccard Similarity and algebraic distance on the other hand are designed to prefer intra-cluster edges which can also be seen in our experiments. On the sparsified networks this has the effect that the Louvain algorithm detects communities that are different from the communities it detects on the original network. On synthetic networks, local filtering with these methods preserves and even enforces the community structure. They are able to preserve the ground truth communities up to a ratio of kept edges of 0.15. This suggests that Simmelian Backbones, Jaccard Similarity and algebraic distance with local filtering indeed keep and enforce some community structure but that on networks without clearly detectable community structure this is not necessarily the same structure as the structure that is found by the Louvain algorithm.

7.5.4 Running Time

Measured running times are shown in Figure 7.11. Random Edge sparsification is clearly the fastest method, closely followed by Local Degree. Jaccard Similarity is also not much slower and scales also very well. Therefore, these methods are well suited for large-scale networks in the range of millions to billions of edges. The efficiency of the Jaccard Similarity method shows that our parallel triangle counting implementation is indeed very scalable. The authors also proposed inexact Jaccard coefficient calculation [SPR11] for a further speedup though given our numbers it can be doubted if – given an efficient triangle counting algorithm – this is necessary or helpful at all. Algebraic distance is a bit slower but scales very well nevertheless. Using fewer systems or iterations could further speed-up algebraic distance if speed is an issue. Both Simmelian methods are significantly slower than the other methods, but still efficient enough for the network sizes we consider. The visible difference between quadrilateral and triangular Simmelian Backbones can be explained by the difference between triangle and quadrangle counting, additionally we did not parallelize the latter. While the time complexity in O-notation of Edge Forest Fire is difficult to assess, it seems to be slightly faster than Simmelian Backbones.

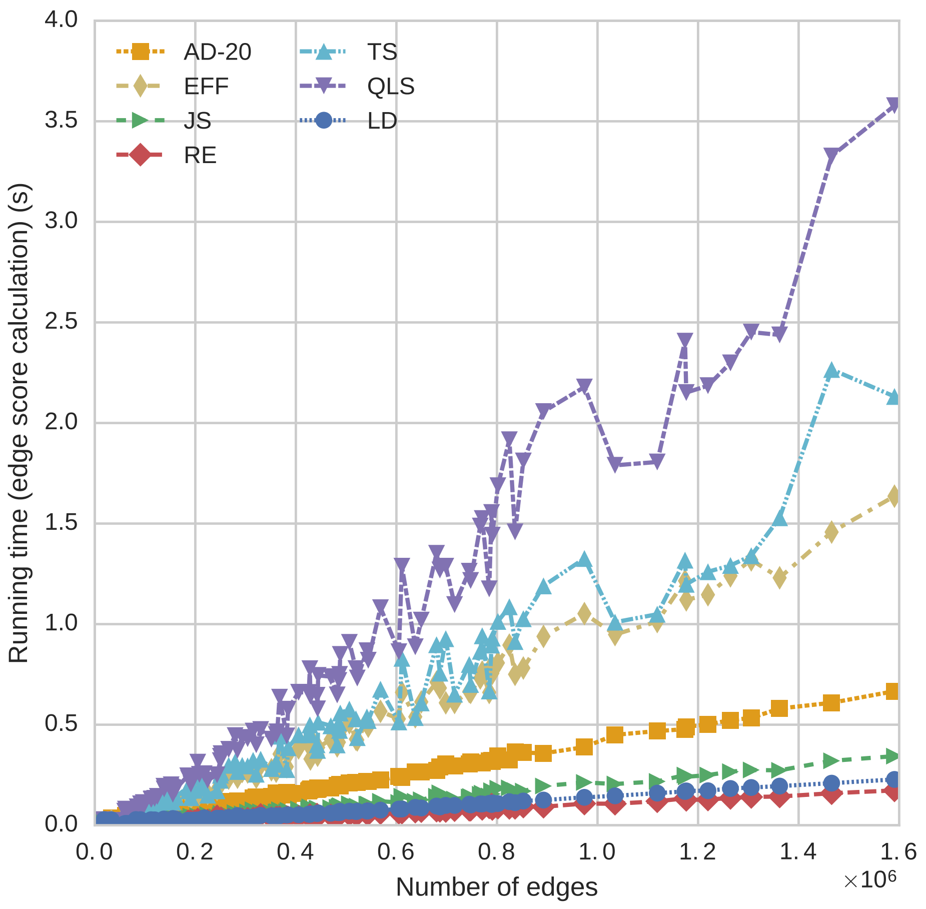


Figure 7.11: Running times of various edge scoring methods on the Facebook networks

7.6 Conclusion

Our experimental study on networks from Facebook, Twitter, Google+ and LiveJournal as well as synthentically generated networks shows that several sparsification methods are capable of preserving a set of relevant properties of social networks when up to 80% of edges have been removed.

Random edge deletion performs surprisingly well and retains the community structure on the studied social networks. We propose local filtering as a generally applicable and computationally cheap post-processing step for edge sparsification methods that improves the preservation of almost all properties as it leads to a more equal rate of filtering across the network. Simmelian Backbones, Jaccard Similarity and algebraic distance prefer intra-cluster edges and thus do not keep global structures but with the added local filtering step they are able to enforce and retain a community structure as it was already shown for Jaccard Similarity. However, the preserved community structure is not necessarily the same as the one the Louvain algorithm finds. Our novel method Local Degree, which is based on the notion that connections to hubs are highly important for the network's structure, in contrast preserves shortest paths and the overall connectivity of the network. This can be seen at the almost perfectly preserved diameter. For preserving the community structure, it is unsuitable though. Our adaptation of the Forest Fire sampling algorithm to edge scoring depends strongly on the specific network's structure but seems not better than random edge sampling for preserving the community structure.

We hope that the conceptual framework of edge scoring and filtering as well as our evaluation methods are steps towards a more unified perspective on a variety of related methods that have been proposed in different contexts. Future developments can be easily carried out within this framework and based on our implementations, which are available as part of the open-source network analysis package NetworKit [SSM16].

Many real-world social networks are not static, but evolve over time. We are not aware of any work that has adapted any of the presented methods for dynamic networks. An interesting direction for future work would thus be to adapt and compare sparsification methods for dynamic networks.

In terms of scalability and networks that do not fit into the main memory anymore, variants that work with external memory or in distributed settings could be developed. While basic building blocks such as triangle counting exist [HTC14], the actual sparsification algorithms would need to be adapted.

Regarding scalable community detection, random edge filtering offers an interesting possibility. While reading a graph, edges could be filtered directly and thus only memory for the sparsified graph would be required. As our study shows, 40 to 60% of the edges suffice on the tested instances to detect similar communities. While this might not be a huge reduction, being able to process a twice as large graph might make a difference in practice.

Part III

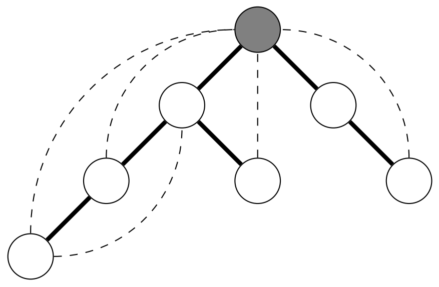
Editing

8 Introduction to Editing

Parts of this chapter are based on the introductions of our papers on quasi-threshold editing [Bra+15b; Got+20b]. The first is joint work with Ulrik Brandes, Ben Strasser and Dorothea Wagner while the second is joint work with Lars Gottesbüren, Philipp Schoch, Ben Strasser, Dorothea Wagner and Sven Zühlsdorf. As they address two variants of the same problem, this chapter introduces both Chapter 9 and 10 that are based on these papers.

The intuitive definition of a community as being internally densely and externally sparsely connected means that the best community would be a clique that is not connected to any other node. A graph with the ideal community structure is thus a disjoint union of cliques. If a graph is not a disjoint union of cliques, we can ask for a set of edge insertion and deletion operations that transform a given graph into a disjoint union of cliques. Finding such a set of editing operations of minium size is commonly called *cluster editing* [BB13]. This can be extended to settings where we are given a cost for each insertion and deletion operation. We then want to minimize the sum of the costs of the needed operations to transform a given graph into a disjoint union of cliques. This is useful for example if we assume that all edges between nodes insides a community should exist, but due to measurement errors edges might not exist in our data. We can then assign costs to express how certain we are that we correctly measured for example an interaction between two proteins or that there was no interaction. In this thesis, we only consider editing problems with uniform costs, an extension of our results to non-uniform costs is part of our ongoing research.

Such an ideal model of a community where each node is connected to every other node is not always realistic. Even in protein interaction networks, this is usually too strict and instead a model with a core and some attached nodes is considered to be more realistic [BHK15]. Nastos and Gao [NG13] consider the class of quasi-threshold graphs as a model for social networks. Quasi-Threshold graphs, also known as *trivially perfect* graphs, can be described as the graphs that are the transitive closure of a rooted forest [Wol65], i.e., every branch from a leaf to the root becomes a clique as illustrated in Figure 8.1a. This means that every connected subgraph has a node that is connected to all other nodes. Such a forest can be seen as a skeleton of the graph. An inductive characterization exists, too: single nodes are quasi-threshold graphs, disjoint unions of quasi-threshold graphs are quasi-threshold graphs and adding a node to a quasi-threshold graph that is connected to all other nodes yields a quasi-threshold graph [YCC96]. Nastos and Gao also consider the quasi-threshold editing problem to transform a given graph into a closest quasi-threshold graph using edge insertions and deletions. Figure 8.1b shows an example. The connected components of these quasi-threshold graphs are then considered as communities. Each community thus consists of a set overlapping cliques.


(a) Quasi-threshold graph with thick skeleton, grey root and dashed transitive closure.

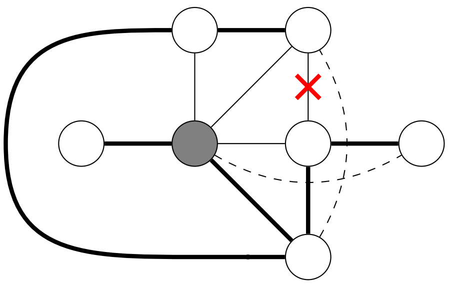
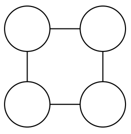
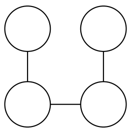
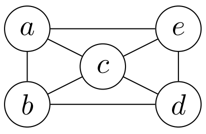

(b) Edit example with solid input edges, dashed inserted edges, a crossed deleted edge, a thick skeleton with grey root.

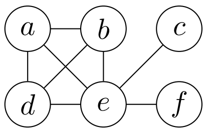
Figure 8.1: Quasi-Threshold graph examples.


(a) A C_4.


(b) A P_4.



(c) A graph with node-induced C_4.



(d) A $\{C_4, P_4\}$-free graph.

Figure 8.2: A C_4, a P_4 and examples for graphs that are (not) $\{C_4, P_4\}$-free.

Quasi-threshold graphs are also exactly the graphs where for every pair of connected nodes u, v, the closed neighborhood of u, $N(u) \cap \{u\}$, is a subset of the closed neighborhood of v, $N(v) \cap \{v\}$, or vice-versa. This naturally defines a hierarchy within each community that might model a social hierarchy or a hierarchy in an organizational structure.

A graph H is an *induced subgraph* of a graph G if there exists an injective mapping π from the nodes V_H of H to the nodes of G such that for two nodes u_i, $u_j \in V_H$, $\{u_i, u_j\}$ is an edge in H if and only if $\{\pi(u_i), \pi(u_j)\}$ is an edge in G. If a graph does not contain a forbidden subgraph H as an induced subgraph, it is called *H-free*. This can be extended to a set of forbidden subgraphs $\mathcal{F}$ where a graph is $\mathcal{F}$-free if it is H-free for all $H \in \mathcal{F}$. Let P_ℓ be a path graph with ℓ nodes. Similarly, let C_ℓ denote a cycle graph with ℓ nodes. Figures 8.2a and 8.2b depict a C_4 and a P_4. The graph depicted in Figure 8.2c is not $\{C_4, P_4\}$-free as the nodes (a, b, d, e) form an induced C_4. In contrast, the graph depicted in Figure 8.2d is $\{C_4, P_4\}$-free. The nodes (a, b, e, c) form a P_4, however, as there is an edge between a and e, the subgraph is not an induced subgraph.

Graphs that are the disjoint union of cliques are exactly the graphs that are P_3-free. Similarly, quasi-threshold graphs are exactly the $\{C_4, P_4\}$-free graphs [YCC96]. Different generalizations like P_4-free deletion [Jia+15] or P_5-free editing [Boh15] have also been considered for community detection.

8.1 Related Work

For quasi-threshold graphs, linear-time algorithms have been proposed for the recognition problem, i.e., if a given graph is a quasi-threshold graph [YCC96; Chu08]. Both construct a skeleton forest if the graph is a quasi-threshold graph. Further, [Chu08] also finds a C_4 or P_4 if there is any and thus the graph is no quasi-threshold graph.

For many choices of $\mathcal{F}$, $\mathcal{F}$-free editing is NP-hard, in particular for $\mathcal{F} = \{C_4, P_4\}$ [NG13]. The $\mathcal{F}$-free edge editing problem for a finite set of finite forbidden subgraphs $\mathcal{F}$ is fixed-parameter tractable (FPT) in the number of edits k [Cai96]. The proof implies a simple branch-and-bound algorithm that we introduce in Section 10.4. For quasi-threshold graphs, it has a running time of $\mathcal{O}(6^k \cdot (n+m))$. For the special case of $\{C_4, P_4\}$-free edge deletion, where only edge deletion operations are allowed, optimized branching rules have been proposed that reduce the running time of the trivial algorithm from $O(4^k \cdot (n+m))$ to $O(2.42^k \cdot (n+m))$ [Liu+15]. To the best of our knowledge, for $\{C_4, P_4\}$-free editing, no improved branching rules have been proposed so far. Further, a polynomial kernel of size $\mathcal{O}(k^7)$ has been introduced for quasi-threshold graphs [DP17]. Unfortunately, as we show in our experiments in Section 9.6, many social networks require more than $m/4$ edits. Further, we show lower bounds of up to millions of edits. Thus, we cannot hope that exact algorithms that are superpolynomial in the number of edits will scale to these graphs. The only heuristic quasi-threshold editing algorithm we are aware of has been proposed by Nastos and Gao [NG13]. Its running time is at least quadratic in the number of nodes and thus still unsuitable for graphs with millions of nodes.

The $\mathcal{F}$-free editing problem also admits a simple formulation as integer liner program (ILP). For cluster editing, a linear programming formulation with cutting planes that are incrementally added (in batches of a few hundred constraints) has been proposed [GW89]. Later, exact algorithms based on integer linear programming as well as kernelization and more efficient FPT algorithms have been considered [BBK11]. In [HH15], the authors combine the FPT algorithm with kernelization as well as upper and lower bounds. Editing to $\{P_4\}$-free graphs has been considered in phylogenomics [Hel+15] using a simple ILP-based approach.

In a bachelor thesis [Boh15], $\{P_5\}$-free editing has been considered for community detection. They apply lower bounds, data reduction rules and rules for disallowing certain edits.

8.2 Our Contribution

In this thesis we primarily consider the quasi-threshold editing problem, though some of our results can be generalized to other editing problems. Our goal is to make it possible to determine whether quasi-threshold editing is a useful community detection algorithm. In Chapter 9, we introduce quasi-threshold mover (QTM), a fast editing heuristic that is close to linear in practice and allows evaluating quasi-threshold editing on large graphs. In the following Chapter 10, we consider solving the quasi-threshold editing problem exactly. The purpose of this is two-fold: First, for small graphs, it allows us to show that our heuristic QTM frequently computes exact solutions or is close to them. Second, the branch-and-bound algorithm we consider also allows to list all solutions exactly once. This allows to analyze the suitability of quasi-threshold editing for community detection in more detail. In a first analysis, we show that even for small social networks there up to several thousands of solutions that also yield different communities.

9 Fast Quasi-Threshold Editing

This chapter is based on joint work with Ulrik Brandes, Ben Strasser and Dorothea Wagner [Bra+15b]. We thank James Nastos for helpful discussions. Compared to the publication, the introduction has been shortened, the appendix of the extended version [Bra+15a] has been revised and integrated, and the lower bounds have been replaced by an improved bound that we originally used in our work on exact editing in Chapter 10.

9.1 Introduction

Our contribution in this chapter are algorithms for the quasi-threshold editing problem that are applicable to graphs with millions of nodes and edges. We present Quasi-Threshold Mover (QTM), a scalable quasi-threshold editing algorithm. Given a graph it computes a quasi-threshold graph which is close in terms of edit count, but not necessarily closest as this problem is NP-hard. We provide an extensive experimental evaluation on synthetic graphs as well as social network and web graphs. We further propose a simplified certifying quasi-threshold recognition algorithm. QTM works in two phases: An initial skeleton forest is constructed by a variant of our recognition algorithm, and then refined by moving one node at a time to reduce the number of edits required. The running time of the first phase is dominated by the time needed to count the number of triangles per edge. The currently best triangle counting algorithms run in $\mathcal{O}(m \cdot \alpha(G))$ [CN85; OB14] time, where $\alpha(G)$ is the arboricity. These algorithms are efficient and scalable in practice on the considered graphs. One round of the second phase needs $\mathcal{O}(n + m \log \Delta)$ time, where Δ is the maximum degree. We show that four rounds are enough to achieve good results.

9.1.1 Outline

This chapter is organized as follows: We begin by describing how we compute lower bounds on the number of edits. We then describe the editing algorithm introduced by Nastos and Gao. In Section 9.4, we introduce the simplified recognition algorithm and the computation of the initial skeleton. The main algorithm is described in Section 9.5. The remainder of the chapter is dedicated to the experimental evaluation.

9.1.2 Preliminaries

We use the notation introduced in Section 1.2. For a skeleton forest, we denote by $p(u)$ the parent of a node u and by T_u the subtree rooted at u.

9.2 Lower Bounds

We propose to compute a simple lower bound. This allows us to show that many of the graphs we consider actually require a large number of edits, meaning that exact algorithms that are superpolynomial in k are infeasible. If the lower bound matches the number of edits needed by the heuristic solution, this shows that the heuristic solution is in fact exact. If the bound is close to the heuristic solution, we can at least be sure that our heuristic solution is not far from optimal.

To edit a graph we must destroy all forbidden subgraphs H. For quasi-threshold editing H is either a P_4 or a C_4. This leads to the following basic algorithm: Find forbidden subgraph H, increase the lower bound, remove all nodes of H, repeat. This is correct as at least one edit incident to H is necessary. If multiple edits are needed then accounting only for one is a lower bound. We can optimize this algorithm by considering node pairs instead of nodes. If we mark all node pairs of H and forbid further subgraphs from using them, the algorithm is still correct as editing one node pair does not influence other node pairs. Further, if H is a P_4 of the form $a - b - c - d$, we do not need to mark $a - d$, as editing $a - d$ will only transform it into a C_4 and thus one of the other node pairs must be edited. Similarly, for a C_4, we can omit one edge. Such lower bounds based on a packing have already been used for cluster editing [HH15] and P_5-free editing [Boh15]. This lower bound corresponds to the *basic* bound that we describe in more detail in Section 10.4.3 in the following chapter on exact editing.

H can be found using the recognition algorithm. However, the resulting running time of $\mathcal{O}(k(n + m))$ does not scale to the large graphs. Instead, we use a listing algorithm that lists all forbidden subgraphs. We iterate over all edges $\{u_0, u_1\}$. If $\{u_0, u_1\}$ is not marked yet, for $u_i, i \in \{0, 1\}$ we iterate over $N(u_i)$ until we find a neighbor v_i such that v_i is not a neighbor of u_{1-i} and neither $\{u_0, v_i\}$ nor $\{u_1, v_i\}$ is marked. If such a neighbor exists for both u_0 and u_1, we found a new forbidden subgraph. We then mark the node pairs $\{u_0, u_1\}$, $\{u_0, v_0\}$, $\{u_0, v_1\}$, $\{u_1, v_0\}$ and $\{u_1, v_1\}$, increase our counter by one and continue. We store the neighbors of every node as well as the marked node pairs in a hash set to ensure that for each edge we only spend $\mathcal{O}(\max\{\deg(u_0), \deg(u_1)\})$ time in expectation. The total running time is thus $\mathcal{O}(m \cdot \Delta)$ in expectation. To improve the running time in practice, we iterate first over the neighbors of the lower-degree node of the edge as we can stop if it has no suitable neighbor.

9.3 The Algorithm proposed by Nastos and Gao

Nastos and Gao [NG13] describe that in their greedy algorithm they test each possible edge addition and deletion (i.e. all $\mathcal{O}(n^2)$ possibilities) in order to choose the edit that results in the largest improvement, i.e. the highest decrease of the number of induced P_4 and C_4. After executing this greedy heuristic they revert the last few edits and execute a bounded search tree algorithm. If this results in a solution with fewer edits, they repeat this last step until no improvement is possible anymore. We revert ten edits in each step. As they do not provide any algorithmic details, we describe how we implemented the algorithm below.

The main question is how to select the next edit. As far as we know, it is an open problem if it is possible to determine the edit that destroys most P_4 and C_4 in time $o(n^2)$. Therefore, we use the obvious approach that was also implied by Nastos and Gao: execute each possible edit and see how the number of P_4 and C_4 changes. The main ingredient is thus a fast update algorithm for this counter.

As far as we are aware the fastest update algorithm for counting node-induced P_4 and C_4 subgraphs needs amortized time $\mathcal{O}(h^2)$ for each update where h is the h-index of the graph [Epp+12]. While the worst-case bound of the h-index is $\sqrt{m}$ it has been shown that many real-world social networks have a much lower h-index [ES09]. However, this algorithm requires to store many counts for node pairs and triples, which makes it practicability questionable.

Instead, we use a simple algorithm that requires only $\mathcal{O}(n)$ additional memory during updates and needs to store only the counter between updates. In the worst case, it needs $\mathcal{O}(m)$ time for an update. The main idea is to examine the neighborhood structure of the edge to be deleted or inserted. We mark common and exclusive neighbors of the two incident nodes and for each of these neighbors, we iterate over its neighbors. Based on which of them are marked, we can determine how many P_4 and C_4 were destroyed or created by editing the edge.

While we could execute this algorithm m times for the initialization, we instead use a simpler algorithm that needs $\mathcal{O}(m^2)$ time, too. For each edge $\{u, v\}$, we determine the neighbors exclusive to one of them. The product of the sizes of the exclusive neighborhoods gives us the number of P_4 where $\{u, v\}$ is the central edge plus the number of C_4 $\{u, v\}$ is part of. As we count each C_4 four times, we need to determine how many C_4 there are to correct for this. For this, we iterate over the neighbors of u and for each neighbor $u_n \in N(u)$, we iterate over the neighbors of u_n and count how many of them are marked as exclusive neighbor of v. When the graph is a quasi-threshold graph, one of the two sets of exclusive neighbors is always empty. This allows to skip iterating over the neighbors of u_n, reducing the running time to $\mathcal{O}(m \cdot \Delta)$.

Overall, the running time is $\mathcal{O}(m^2 + k \cdot n^2 \cdot m)$. The time needed for the initial counting (first term) is dominated by the time needed for each edit (second term).

9.4 Linear Recognition and Initial Editing

The first linear time recognition algorithm for quasi-threshold graphs was proposed in [YCC96]. In [Chu08], a linear time certifying recognition algorithm based on lexicographic breadth first search was presented. However, as the authors note, sorted node partitions and linked lists are needed, which incur large constant factors in the running time. We simplify their algorithm to only require arrays but still provide negative and positive certificates. Further, we only need to sort the nodes once to iterate over them by decreasing degree. Our algorithm constructs the forest skeleton of a graph G. If it succeeds G is a quasi-threshold graph and outputs for each node v a parent node $p(v)$. If it fails it outputs a forbidden subgraph H.

To simplify our algorithm we start by adding a super node r to G that is connected to every node and obtain G'. G is a quasi-threshold graph if and only if G' is one. As G' is connected, its skeleton is a tree. A core observation is that higher nodes in the tree must have higher degrees, i.e., $\deg(v) \leq \deg(p(v))$. We therefore know that r must be the root of the tree. Initially, we set $p(u) = r$ for every node u. We process all remaining nodes ordered decreasingly by degree. Once a node is processed its position in the tree is fixed. Denote by u the node that should be processed next. We iterate over all non-processed neighbors v of u and check whether $p(u) = p(v)$ holds and afterwards set $p(v)$ to u. If $p(u) = p(v)$ never fails then G is a quasi-threshold graph as for every node x (except r) we have that by construction the neighborhood of x is a subset of the one of $p(x)$. If $p(u) \neq p(v)$ holds at some point then a forbidden subgraph H exists. Either $p(u)$ or $p(v)$ was processed first. Assume without loss of generality that it was $p(v)$. We know that no edge $\{v, p(u)\}$ can exist because otherwise $p(u)$ would have assigned itself as parent of v when it was processed. Further, we know that $p(u)$'s degree can not be smaller than u's degree as $p(u)$ was processed before u. As v is a neighbor of u we know that another node x must exist that is a neighbor of $p(u)$ but not of u, i.e., $\{u, x\}$ does not exist. The subgraph H induced by the 4-chain $v - u - p(u) - x$ is thus a P_4 or C_4 depending on whether the edge $\{v, x\}$ exists. We have that u, v and $p(u)$ are not r as $p(v)$ was processed before them and r was processed first. As x has been chosen such that $\{u, x\}$ does not exist but $\{u, r\}$ exist $x \neq r$. H therefore does not use r and is contained in G.

From Recognition to Editing. We modify the recognition algorithm to construct a skeleton for arbitrary graphs. This skeleton induces a quasi-threshold graph Q. We want to minimize Q's distance to G. Note that all edits are performed implicitly, we do not actually modify the input graph for efficiency reasons. The only difference between our recognition and our editing algorithm is what happens when we process a node u that has a non-processed neighbor v with $p(u) \neq p(v)$. The recognition algorithm constructs a forbidden subgraph H, while the editing algorithm tries to resolve the problem. We have three options for resolving the problem: we ignore the edge $\{u, v\}$, we set $p(v)$ to $p(u)$, or we set $p(u)$ to $p(v)$. The last option differs from the first two as it affects all neighbors of u. The first two options are the decision if we want to make v a child of u even though $p(u) \neq p(v)$ or if we want to ignore this potential child. The first option implies deleting $\{u, v\}$. The second option implies that all edges from v to its ancestors are deleted and edges to u's ancestors are inserted. Parts of these ancestors might be shared and thus in fact not cause any edits. The third option means that all edges from u to its ancestors are deleted and new edges to v's ancestors are inserted. The same also applies to all neighbors that u keeps, for all other neighbors the edge to the neighbor is deleted. Again, parts of the ancestors might be shared and thus not cause any edits.

We start by determining a preliminary set of children by deciding for each non-processed neighbor of u whether we want to keep or discard it. These preliminary children elect a new parent by majority. We set $p(u)$ to this new parent. Changing u's parent can change which neighbors are kept. We therefore reevaluate all the decisions and obtain a final set of children for which we set u as parent. Then the algorithm simply continues with the

```
    Input: $G = (V, E)$
    Output: Parent assignment $p$ for each node
 1  Sort $V$ by degree in descending order using bucket sort;
 2  $p : V \to V \cup \{\emptyset\}, u \mapsto \emptyset$;
 3  depth : $V \to \mathbb{N}_0, u \mapsto 0$;
 4  Count triangles $t(\{u, v\})$ and calculate $p_c(\{u, v\})$;
 5  foreach $u \in V$ do
        // Process node $u$
 6      $N \leftarrow \{v \in N(u) \,|\, v$ not processed and $p(u) = p(v)$ or $(p_c(\{u, v\}) \leq p_c(\{v, p(v)\})$
          and depth$(v) \leq t(\{u, v\}) + 1)\}$;
 7      $p_n \leftarrow$ the most frequent value of $p(x)$ for $x \in N$;
 8      if $p_n \neq p(u)$ then
 9          $p(u) \leftarrow p_n$;
10          depth$(u) \leftarrow 0$;
11          $p_c(\{u, p_n\}) \leftarrow \infty$;
12      foreach $v \in N(u)$ that has not been processed do
13          if $p(u) = p(v)$ or $(p_c(\{u, v\}) < p_c(\{v, p(v)\})$ and depth$(v) < t(\{u, v\}) + 1)$
              then
14              $p(v) \leftarrow u$;
15              depth$(v) \leftarrow$ depth$(v) + 1$;
```

Algorithm 9.1: The Initialization Algorithm

next node.

What remains to describe is when our algorithm keeps a potential child. It does this using two edge measures: The number of triangles $t(e)$ in which an edge e participates and a pseudo-C_4-P_4-counter $p_c(e)$, which is the sum of the number of C_4 in which e participates and the number of P_4 in which e participates as central edge. Computing $p_c(x, y)$ is easy given the number of triangles and the degrees of x and y as $p_c(\{x, y\}) = (\deg(x) - 1 - t(\{x, y\})) \cdot (\deg(y) - 1 - t(\{x, y\}))$ holds. Having a high $p_c(e)$ makes it likely that e should be deleted. We keep a potential child only if two conditions hold. The first is based on triangles. We know by construction that both u and v have many edges in G towards their current ancestors. Keeping v is thus only useful if u and v share a large number of ancestors as otherwise the number of induced edits is too high. Each common ancestor of u and v results in a triangle involving the edge $\{u, v\}$ in Q. Many of these triangles should also be contained in G. We therefore count the triangles of $\{u, v\}$ in G and check whether there are at least as many triangles as v has ancestors. The other condition uses $p_c(e)$. The decision whether we keep v is in essence the question of whether $\{u, v\}$ or $\{v, p(v)\}$ should be in Q. We only keep v if $p_c(\{u, v\})$ is not higher than $p_c(\{v, p(v)\})$.

In Algorithm 9.1 we provide the full initialization heuristic as pseudo code. Note that while for the parent calculation we use $\leq$ for comparisons we use $<$ for the final selection

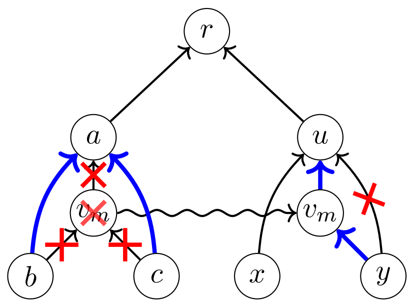


Figure 9.1: Moving v_m example. The drawn edges are in the skeleton and directed towards the root r. By moving v_m, crossed edges are removed and thick blue edges are inserted. The old parent node a becomes the parent of the old children b and c. From the new parent node u, the child x is not adopted while y is.

of the neighbors to keep in order to not to wrongly assign too many neighbors to u.

The time complexity of this editing heuristic is dominated by the triangle counting algorithm as the rest is linear.

9.5 The Quasi-Threshold Mover Algorithm

The Quasi-Threshold Mover (QTM) algorithm iteratively increases the quality of a skeleton T using an algorithm based on local moving. Local moving is a technique that is successfully employed in many heuristic community detection algorithms [Blo+08; GKW14; RN11]. As in most algorithms based on this principle, our algorithm works in rounds. In each round it iterates over all nodes v_m in random order and tries to move v_m. In the context of community detection, a node is moved to a neighboring community such that a certain objective function is increased. In our setting we want to minimize the number of edits needed to transform the input graph G into the quasi-threshold graph Q implicitly defined by T. We need to define the set of allowed moves for v_m in our setting. Moving v_m consists of moving v_m to a different position within T and is illustrated in Figure 9.1. We need to choose a new parent u for v_m. The new parent of v_m's old children is v_m's old parent. Besides choosing the new parent u, we select a set of children of u that are *adopted* by v_m, i.e., their new parent becomes v_m. Among all allowed moves for v_m we choose the move that reduces the number of edits as much as possible. Doing this in sub-quadratic running time is difficult as v_m might be moved anywhere in G. By only considering the neighbors of v_m in G and a constant number of nodes per neighbor in a bottom-up scan in the skeleton, our algorithm has a running time in $\mathcal{O}(n + m \log \Delta)$ per round. While our algorithm is not guaranteed to be optimal as a whole we can prove that for each node v_m we choose a move that reduces the number of edits as much as possible. Our experiments show that given the result of the initialization heuristic our moving algorithm performs well in practice. They further show that in practice four rounds are good enough which results in a near-linear total running time.

9.5.1 Basic Idea

Our algorithm starts by isolating v_m, i.e., removing all incident edges in Q. It then finds a position at which v_m should be inserted in T. If v_m's original position was optimal then it will find this position again. For simplicity, we assume again that we have a virtual root r that is connected to all nodes. Isolating v_m thus means that we move v_m below the root r and do not adopt any children.

Choosing u as parent of v_m requires Q to contain edges from all ancestors of u to v_m. Further if v_m adopts a child c of u then Q must have an edge from every descendant of c to v_m. How good a move is depends on how many of these edges already exist in G and how many edges incident to v_m in G are not in Q. To simplify notation we will refer to the nodes incident to v_m in G as *v_m-neighbors.*

We start by identifying which children a node should adopt. For this we define the *child closeness* $\text{child}_{\text{close}}(c)$ of c as the number of v_m-neighbors in the subtree of c minus the non-v_m-neighbors. A node c is a *close child* if $\text{child}_{\text{close}}(c) > 0$. If v_m chooses a node u as new parent then it should adopt all close children of u, as each of them reduces the number of required edits. A node can only be a close child if it is a neighbor of v_m or if it has a close child. Our algorithm starts by computing all close children and their closeness using many short DFS searches in a bottom up fashion. Knowing which nodes are close children, we can identify which nodes are good parents for v_m. A potential parent must have a close child or must be a neighbor of v_m. Using the set of close children we can easily derive a set of parent candidates and an optimal selection of adopted children for every potential parent. We need to determine the candidate with the fewest edits. We do this in a bottom-up fashion.

With the techniques described in the following, we can implement this algorithm in $\mathcal{O}(n + m \log \Delta)$ per round For every move, we insert $\mathcal{O}(d_G(v_m))$ elements into a priority queue and explore a limited part of the skeleton forest. The running time for this is amortized $\mathcal{O}(d_G(v_m) \log d_G(v_m))$ per move. We analyze the running time complexity using tokens. Initially only the v_m-neighbors have tokens. The tokens are consumed by the short DFS searches and the processing of parent nodes. The details of the analysis are described in Section 9.5.5.

9.5.2 Close Children

To find all close children we start a DFS from every potential close child u that explores u's subtree. A node u is a close child if this DFS finds more v_m-neighbors than non-v_m-neighbors. Unfortunately we can not fully run all these searches as this requires too much running time. Therefore, a DFS is aborted if it finds more non-v_m-neighbors than v_m-neighbors. For every aborted DFS, we store a pointer DFS_{next} that points to the last node visited and allows to continue the DFS. To avoid exploring subtrees twice, a DFS skips already explored subtrees using these pointers. By using a priority queue, we ensure that nodes are explored in a bottom-up fashion and no DFS will be started in an already explored subtree. We exploit that close children are v_m-neighbors or have themselves close children. Initially we fill the priority queue of potential close children

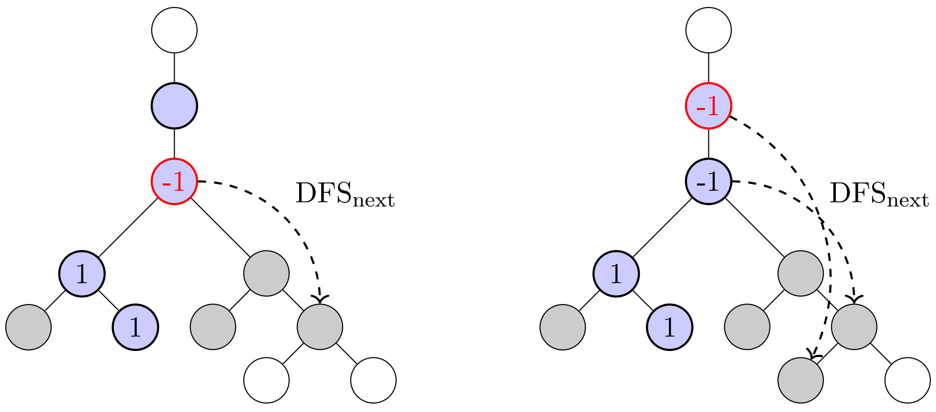


Figure 9.2: Example of two states of the bottom-up search for determining child$_{\text{close}}$ visualized using the skeleton forest, each after processing the node marked in red. Blue nodes are neighbors of v_m, gray nodes have been visited by a DFS. The values in the nodes denote the values of child$_{\text{close}}$. The dashed arrows denote the pointers to the end of the DFS. The two blue nodes in the bottom left have been processed first, both have a child$_{\text{close}}$ of 1. The DFS starting from the red node in the left figure does not descend into the left subtree but directly continues with the right subtree. The DFS from the red node in the right figure follows the dashed arrow from its child to resume its DFS and then visits only one additional node before it stops again and records its state with the second dashed arrow.

with the neighbors of v_m and when a new close child is found we add its parent to the queue. Figure 9.2 shows two examples for the nodes explored by these searches.

Let u denote the current node removed from the queue. We start a DFS from u and if it explores the whole subtree then u is a close child. If not, it stops when child$_{\text{close}}(u)$ reaches -1 and then stores a pointer to the last-visited node. If the DFS of u starts by first inspecting the wrong children then it can get stuck because it would see the v_m-neighbors too late. The DFS must thus first consider the close children of u as close children are exactly the children whose subtrees contain more v_m-neighbors than non-v_m-neighbors. To assure that u knows which children are close, every close child reports itself to its parent. As all children of u have a greater depth than u, the priority queue ensures that they are processed before the DFS of u starts.

Algorithm 9.2 shows this DFS as pseudo code. We start by initializing child$_{\text{close}}(u)$ to the sum of the close children, i.e., those that reported to u and mark u as processed. Then we check if u is a v_m-neighbor and update the score accordingly. We only actually start a DFS if child$_{\text{close}}(u) \geq 0$ as during this DFS the score can only decrease as only subtrees of close children contain more neighbors than non-neighbors and we already considered subtrees of close children. We only continue the DFS into nodes x that have not been processed or whose child$_{\text{close}}(x) < 0$, as all other children are close and have thus already been considered in the score of the parent node. For every visited node, we decrease the

```
$\text{child}_{\text{close}}(u) \leftarrow \sum$ over $\text{child}_{\text{close}}$ of close $u$-children;
mark $u$ as processed;
if $u$ is $v_m$-neighbor then
    $\text{child}_{\text{close}}(u) \leftarrow \text{child}_{\text{close}}(u) + 1$;
else
    $\text{child}_{\text{close}}(u) \leftarrow \text{child}_{\text{close}}(u) - 1$;
if $\text{child}_{\text{close}}(u) \geq 0$ and $u$ has children then // Start a DFS from $u$
    $x \leftarrow$ first child of $u$;
    while $x \neq u$ do
        if $x$ not processed or $\text{child}_{\text{close}}(x) < 0$ then
            $\text{child}_{\text{close}}(u) \leftarrow \text{child}_{\text{close}}(u) - 1$;
            $x \leftarrow \text{DFS}_{\text{next}}(x)$;
            if $\text{child}_{\text{close}}(u) < 0$ then
                $\text{DFS}_{\text{next}}(u) \leftarrow x$;
                break;
            $x \leftarrow$ next node in DFS order after $x$ below $u$;
        else
            // skip subtree of $x$
            $x \leftarrow$ next node in DFS order after the subtree of $x$ below $u$;
```

Algorithm 9.2: The algorithm for determining $\text{child}_{\text{close}}(u)$.

score of u by one. Every node x stores a $\text{DFS}_{\text{next}}(x)$ pointer that initially points to itself and is set to the last visited node when the DFS is aborted, thus indicating the next node that should be visited by a DFS that reaches x. We follow the DFS_{next}-pointer to potentially skip the part of the tree already explored from x and then continue with the next node in DFS order, unless the score of u is now below 0, i.e., -1, in which case we abort the DFS and set $\text{DFS}_{\text{next}}(u)$ to x. This ensures that when another node encounters u, it continues its DFS with x.

After moving v_m, we reset all DFS_{next} pointers, all markers for processed nodes and all $\text{child}_{\text{close}}$ values. As only values of processed nodes are modified, it suffices to initialize the values once and then only reset the values of processed nodes.

9.5.3 Potential Parents

Let X_w be the set of nodes consisting of w, the ancestors of w, the close children of w and the descendants of the close children of w. Moving v_m below w, i.e., choosing w as parent, requires us to insert an edge from v_m to every non-v_m-neighbor in X_w. Likewise, not including v_m-neighbors in X_w requires us to delete an edge for each of them. We therefore want X_w to maximize the number of v_m-neighbors minus the number of non-v_m-neighbors. We compute this score recursively. For this, we restrict the graph to the subtree T_u of a node u and first only consider potential parents and edits within that subgraph. If we

```
1  foreach $v_m$-neighbor $u$ do
2      push $u$;
3  while queue not empty do
4      $u \leftarrow$ pop;
5      determine $\mathrm{child}_{\mathrm{close}}(u)$ by DFS;
6      $x \leftarrow$ max over $\mathrm{score}_{\max}$ of reported $u$-children;
7      $y \leftarrow \sum$ over $\mathrm{child}_{\mathrm{close}}$ of close $u$-children;
8      if $u$ is $v_m$-neighbor then
9          $\mathrm{score}_{\max}(u) \leftarrow \max\{x, y\} + 1$;
10     else
11         $\mathrm{score}_{\max}(u) \leftarrow \max\{x, y\} - 1$;
12     if $\mathrm{child}_{\mathrm{close}}(u) > 0$ or $\mathrm{score}_{\max}(u) > 0$ then
13         report $u$ to $p(u)$;
14         push $p(u)$;
15 Best $v_m$-parent corresponds to $\mathrm{score}_{\max}(r)$;
```
Algorithm 9.3: Pseudo-Code for moving v_m

later consider T_a for a node a that is an ancestor of u, each of these parents will incur the same number of edits in $T_a \setminus T_u$. Thus, we can recursively first choose the best candidate in T_u and then continue with the subtrees of parents of u until we reach T_r and obtain the best candidate for the whole graph.

We denote by $\mathrm{score}_{\max}(u)$ the maximum number of v_m-neighbors minus non-v_m-neighbors in X_w restricted to T_u over all potential parents w in T_u. We determine in a bottom-up fashion all $\mathrm{score}_{\max}(u)$ that are greater than 0. The value in $\mathrm{score}_{\max}(r)$ of the root r then yields the best choice for the whole graph as its "subtree" encompasses the whole graph. As isolating v_m, i.e., choosing r as parent and not adopting any children, gives a score of 0, it suffices to consider nodes with $\mathrm{score}_{\max}(u) > 0$ to determine the best choice. If u itself is the best parent in T_u then the value of $\mathrm{score}_{\max}(u)$ is the sum over the closenesses of all of u's close children ± 1. If the subtree T_c of a child c of u contains the best parent then $\mathrm{score}_{\max}(u) = \mathrm{score}_{\max}(c) \pm 1$. The ± 1 depends on whether u is a v_m-neighbor. Unfortunately, not only potential parents u have a $\mathrm{score}_{\max}(u) > 0$. However, we know that every node u with $\mathrm{score}_{\max}(u) > 0$ is a v_m-neighbor or has a child w with $\mathrm{score}_{\max}(w) > 0$. We can therefore process all $\mathrm{score}_{\max}$ values in a similar bottom-up way using a tree-depth ordered priority queue as we used to compute $\mathrm{child}_{\mathrm{close}}$. As both bottom-up procedures have the same structure we can interweave them as optimization and use only a single queue. The algorithm is illustrated in Algorithm 9.3 in pseudo-code form. Whenever we propagate $\mathrm{score}_{\max}$ to a parent node, we can also propagate which node is the best parent.

After we determined the best parent, we need to determine which children should be attached to v_m. For this, we can simply visit all previously visited nodes (we can store them) and for each node c of them determine if it is a close child of the best parent by testing if $\text{child}_{\text{close}}(c) > 0$ and if the parent of c is the best parent.

9.5.4 Proof of Correctness

In this section, we prove that the algorithm presented in the previous sections is actually correct, i.e., computes the correct parent and children to attach. We start by showing that we process all close children and correctly set their scores.

Proposition 9.1. *Either* $\text{child}_{\text{close}}(u)$ *is the number of neighbors of v_m in the subtree of u minus the number of non-neighbors, or there are more non-neighbors than neighbors in the subtree of u. In the latter case, if u has been processed by Algorithm 9.3, then* $\text{child}_{\text{close}}(u) = -1$ *and* $\text{DFS}_{\text{next}}(u)$ *points to a node in the subtree of u such that -1 is the number of neighbors minus the number of non-neighbors of v_m among all nodes in DFS order between u and* $\text{DFS}_{\text{next}}(u)$*, both included, as well as all nodes in subtrees of close children of u that are in the DFS order after* $\text{DFS}_{\text{next}}(u)$.

Proof. First, we show that all nodes u with $\text{child}_{\text{close}}(u) \geq 0$ are part of the queue and thus processed by Algorithm 9.3. Only neighbors of v_m and their ancestors can have $\text{child}_{\text{close}}(u) \geq 0$. All neighbors of v_m are processed (line 2). For non-neighbors u, $\text{child}_{\text{close}}(u) \geq 0$ means that one of their children c is close, i.e. has $\text{child}_{\text{close}}(c) > 0$. As in this case c inserts $p(c)$ into the queue (line 14) also in this case u will be processed. Therefore, when the algorithm terminates, all nodes u with $\text{child}_{\text{close}}(u) \geq 0$ have been processed.

To prove the claim it thus suffices to show that for a processed node u the value of $\text{child}_{\text{close}}(u)$ is correctly determined by Algorithm 9.2. We do this by structural induction.

We first show that the claim is true if no node below u has been processed. This means that u is a neighbor of v_m, as only neighbors of v_m and parents of processed nodes are inserted into the queue. Then u has no close children, thus initially $\text{child}_{\text{close}}(u) = 1$ after line 4. If u is a leaf, we are finished and $\text{child}_{\text{close}}(u)$ is correctly set to 1. Otherwise, we start the DFS. We are always in the first case of the if-condition as no node below u has been processed. For every node we visit in line 9, we decrease $\text{child}_{\text{close}}(u)$ by 1 in line 11. We only set the DFS_{next} pointer of a node that is processed in line 14, thus all DFS_{next} pointers below u point to themselves, thus line 12 does nothing. If there is only one node below u, our DFS terminates and $\text{child}_{\text{close}}(u)$ is correctly set to 0. Otherwise, we stop the DFS if $\text{child}_{\text{close}}(u) < 0$, i.e., as we always decrement by 1, if $\text{child}_{\text{close}}(u) = -1$ and set $\text{DFS}_{\text{next}}(u)$ to the currently considered node x. As there are only non-neighbors below u, we correctly decremented the counter for all of them including x as claimed.

We now assume that the claim holds for all processed nodes below u. All nodes inserted into the queue have a smaller depth than already processed nodes. Thus, as we showed before, all nodes x below u with $\text{child}_{\text{close}}(x) \geq 0$ have already been processed. In the first lines of Algorithm 9.2, we already consider all scores of close children as well as u itself. If now $\text{child}_{\text{close}}(u) < 0$, it must be -1 as we only subtracted 1 and close children can only contribute positive values. We then correctly skip the DFS as only close children contain more neighbors than non-neighbors, children that are not close can thus never increase the score. The DFS pointer $\text{DFS}_{\text{next}}(u)$ correctly remains at u as this score only considers nodes in the subtrees of close children and u itself. Now consider the case that we start a DFS. Whenever we visit a node x, there are three possible cases:

1. Node x has been processed and $\text{child}_{\text{close}}(x) \geq 0$. In this case, we know that its score is correct by our induction hypothesis. Further, $\text{child}_{\text{close}}(x)$ is either 0 and does not contribute to the score or x is a close child and $\text{child}_{\text{close}}(x)$ has already been considered in the score of its parent, either by our induction hypothesis or because of line 1. Therefore, these nodes are correctly skipped in line 18.

2. Node x has not been processed yet. This means x is a not a neighbor of v_m (otherwise it would have been processed). Thus, we correctly decrease $\text{child}_{\text{close}}(u)$ to account for this non-neighbor. Then we can continue the DFS.

3. Node x has been processed and $\text{child}_{\text{close}}(x) < 0$. In this case we know from the induction hypothesis that $\text{child}_{\text{close}}(x) = -1$ and that this is exactly the number of neighbors minus the number of non-neighbors of v_m of all nodes in DFS order from x up to $\text{DFS}_{\text{next}}(x)$, including both, and all nodes in subtrees of children c of x with $\text{child}_{\text{close}}(c) > -1$, which we ignore anyway in the DFS. We decrease $\text{child}_{\text{close}}(u)$ by 1 which correctly considers the nodes between x and $\text{DFS}_{\text{next}}(x)$ in DFS order, both included. Then we jump to $\text{DFS}_{\text{next}}(x)$ and do not visit $\text{DFS}_{\text{next}}(x)$ but the next node in DFS order which is correct as $\text{DFS}_{\text{next}}(x)$ has already been considered.

When the DFS ends, either we have visited and thus computed the correct score for all descendants of u or the DFS ended with $\text{child}_{\text{close}}(u) = -1$ and we store the location of the last visited node in $\text{DFS}_{\text{next}}(u)$. In the latter case, all nodes up to this point have been considered as we have outlined before. Therefore, the claim is now also true for u. □

In order to not to need to evaluate all nodes as potential parents we make use of the following observation:

Proposition 9.2. *Only nodes with close children and neighbors of v_m need to be considered as parents of v_m.*

Proof. We show this by contradiction. Assume that the best parent u has no close children and is not a neighbor of v_m. Thus attaching children of u to v_m cannot decrease the number of needed edits, so we can assume that no children will be attached. Then choosing $p(u)$ as parent of v_m will save one edit as u is no neighbor of v_m. This is a contradiction to the assumption that u is the best parent. □

So far we have only evaluated edits below nodes and identified all possible parents which are also processed as we have established before. If we want to know for a potential parent u how many edits we can save by moving some of its children to v_m this is the sum of $\text{child}_{\text{close}}(c)$ for all close children of u. The part that is still missing is the evaluation of the edits above a potential parent u.

Theorem 9.3. *Consider the subtree T_u of a node u. Then for the subgraph of T_u, either* $\text{score}_{\max}(u) < 0$ *or u has been processed and* $\text{score}_{\max}(u)$ *is correctly computed by Algorithm 9.3.*

Proof. First, we show that u is processed if $\text{score}_{\max}(u) \geq 0$. There are two possibilities: Either u is the best parent or we a node below u is the best parent. In the first case by Proposition 9.2 either u is a neighbor of v_m or u has a close child which means that u is processed. In the second case, let b be the best parent in T_u, more precisely, b is a node that has $\text{score}_{\max}(u)$ v_m-neighbors minus non-v_m-neighbors in X_b restricted to T_u and is closest to u, i.e., there is no ancestor of b in T_u that has the same score. Again, by Proposition 9.2, we know that b is processed. Then the parent $p(b)$ is processed and $\text{score}_{\max}(p(b)) \geq \text{score}_{\max}(b) - 1$ as $p(b)$ has at most one non-v_m-neighbor more than b. This continues until we either reach u and we are done, or $\text{score}_{\max}(a) = 0$ for some node on the path between b and u, $a \neq b$ and $a \neq u$. However, then $p(a)$ is at least as good as a parent as b, as X_b contains as many v_m-neighbors as non-v_m-neighbors in T_a as indicated by $\text{score}_{\max}(a) = 0$ and thus the score of $p(a)$ is at least as good. This is a contradiction to b being the best parent closest to u and thus shows that there is no node with $\text{score}_{\max}(a) = 0$ on the path between b and u. This means all nodes between b and u are processed.

The proof for the correctness of the calculation of $\text{score}_{\max}$ is given by structural induction on the processed nodes of the tree skeleton. We start with the initial step which is a node u below which no node has been processed. This means u is a neighbor of v_m. Further, no nodes reported to u and u has no close children, thus x and y are 0. Thus, $\text{score}_{\max}(u) = 1$ which is correct as u as sole neighbor of v_m in u is the best parent and saves one edit.

For the induction we can assume that the theorem holds for all processed children of u. If u is a v_m-neighbor or not has the same influence for all nodes we could choose in T_u. Therefore, we do not need to reconsider any decisions that were made below u. The only decision is thus whether u is the best parent or if the best parent in the subtree of a child c of u is the best parent. In the first case, the sum of $\text{child}_{\text{close}}$ of all close children of u is the number of neighbors minus non-neighbors of v_m we need to consider apart from u. In the second case, the maximum over all $\text{score}_{\max}(c)$ for all reported children c of u gives us the number of neighbors minus non-neighbors apart from u. Thus, picking the maximum of x and y in lines 9 and 11 is correct. In the second case we choose the candidate of the child that had the maximum score. We then correctly add or subtract 1 from this score depending on whether u is a v_m-neighbor or not, as this saves one edit or requires an additional edit. □

As T_r is the whole graph and r is processed as a neighbor of v_m, the best solution of the whole graph is available at r. Therefore, the QTM algorithm optimally solves the problem of finding a new parent and a set of its children that shall be adopted.

9.5.5 Proof of the Running Time

After showing the correctness of the algorithm, we will now show that the running time is indeed $\mathcal{O}(m \log(\Delta))$ per iteration and amortized $\mathcal{O}(d \log(d))$ per node.

During the whole algorithm we maintain a depth value for each node that specifies the depth in the forest at which the node is located. Whenever we move a node, we update these depth values. This involves decreasing the depth values of all descendants of the node in its original position and increasing the depth values of all descendants of the node at the new position. Unfortunately it is not obvious that this is possible in the claimed running time as a node v_m might have more than $\mathcal{O}(\deg(v_m))$ descendants.

Note that a node is adjacent to all its descendants and ancestors in the edited graph. This means that every ancestor or descendant that is not adjacent to the node causes an insert. Therefore, v_m must be a neighbor of at least half of the ancestors and descendants at the target position as otherwise choosing r as parent would have been better. This means that updating the depth values at the destination is possible in $\mathcal{O}(d)$ time.

For the update of the values in the original position we need a different, more complicated argument. First of all we assume that initially the total number of edits never exceeds the number of edges as otherwise we could simply delete all edges and get fewer edits[1]. For amortizing the number of needed edits of nodes that have more descendants and ancestors than their degree we give each node tokens for all their neighbors in the edited graph. As the number of edits is at most m the number of initially distributed tokens is in $\mathcal{O}(m)$. Whenever we move a node v_m, it generates tokens for all its new neighbors and itself, i.e. in total at most $2 \cdot \deg(v_m)$ tokens. Therefore, a node has always a token for each of its ancestors and descendants and can use these tokens to account for updating the depth of its previous descendants. In each round only $\mathcal{O}(m)$ tokens are generated, therefore updating the depth values of a node is in amortized time $\mathcal{O}(d)$ per node and $\mathcal{O}(m)$ per iteration.

Using the same argument we can also account for the time that is needed for updating the pointers of each node to its parents and children and for counting the number of initially needed or saved edits.

What we have shown so far means that once we know the best destination we can move a node and update all depth values in time $\mathcal{O}(d)$ amortized over an iteration where all nodes are moved.

The remaining claim is that we can determine the new parent and the new children in time $\mathcal{O}(d \log(d))$ per node. More precisely we will show that only $\mathcal{O}(d)$ nodes are inserted in the queue and we need amortized constant time for processing a node. A standard max-heap that needs $\mathcal{O}(\log(n))$ time per operation can be used for the implementation of the queue.

We maintain scores per node, in particular $\text{child}_{\text{close}}$, $\text{score}_{\text{max}}$, DFS_{next} and a flag if a node was processed. As we only set them for processed nodes, it suffices to initialize them once and then reset them for processed nodes. Thus, they do not require any additional

[1]Our experiments show that this is not true when using the initialization heuristic and thus the running time in practice might be higher in the first round when using the initialization heuristic.

running time.

The basic idea of the main proof is that each neighbor of v_m gets two tokens that it can use to visit descendants or distribute to ancestors that can use those tokens themselves. This is represented by the fact that we increase $\text{child}_{\text{close}}(u)$ and $\text{score}_{\max}(u)$ by 1 if u is a neighbor of v_m. All nodes that are processed are neighbors of v_m or have a child c with $\text{child}_{\text{close}}(c) > 0$ or $\text{score}_{\max}(c) > 0$. When we process a node u that is no v_m-neighbor, we consume tokens that are passed to u from close children and from children with $\text{score}_{\max}(c) > 0$. At the end the rest of the tokens is passed to the parent.

First we consider Algorithm 9.3 and ignore the computation of $\text{child}_{\text{close}}(u)$. For all processed nodes u, $\text{child}_{\text{close}}(u) \leq \text{score}_{\max}(u)$. This is because in line 9 and 11, $\text{score}_{\max}(u)$ is set to the maximum of x and y plus/minus 1. However, y plus/minus 1 is also the initialization of $\text{child}_{\text{close}}(u)$, and after this initialization, $\text{child}_{\text{close}}(u)$ is only decreased. Thus, the sum of $\text{score}_{\max}(c)$ for all reported u-children is an upper bound for y. Regardless if $x > y$ or not, we can therefore always attribute the tokens in $\text{score}_{\max}(u)$ and the token consumed in line 11 to tokens that have been passed to u as tokens from $\text{score}_{\max}(c)$ for some children c of u.

Next, we consider the computation of $\text{child}_{\text{close}}(u)$ in Algorithm 9.2. If u has no close children and is not a v_m neighbor, we do not start a DFS and attribute the constant amount of work done in Algorithm 9.2 to the token consumed in Algorithm 9.3. If u is no v_m-neighbor, we consume a token for processing u in line 6. Apart from the DFS only constant work is done per node, so consuming one token is enough for that.

For each visited node in the DFS only a constant amount of work is needed as traversing the tree, i.e. we attribute possibly traversing a node multiple times to find the next node to the first visit. Without keeping a stack this needs a tree structure where we can determine the next child c' after a child c of a node u in constant time. This can be implemented by storing in node c the position of c in the array (or list) of children in $p(c)$. This also allows deleting entries in the children list in constant time (in an array deletion can be implemented as swap with the last child).

Whenever we visit a node that has not been processed yet or that has $\text{child}_{\text{close}}(x) < 0$, we consume one token of $\text{child}_{\text{close}}(u)$. When this is not the case, i.e. $\text{child}_{\text{close}}(x) > -1$ and we thus skip x, x has been processed already and we account our visiting of x to the processing of x. This is okay as we skip each node only once during a DFS: After the DFS starting at u has finished, either $\text{child}_{\text{close}}(u) > -1$ and an upcoming DFS will not descend into the subtree of u anymore or the DFS stopped in line 13 and thus $\text{DFS}_{\text{next}}(u)$ has been set to the last visited node (which is not a skipped node). In the latter case, when we visit u in an upcoming DFS, this DFS will directly jump to $\text{DFS}_{\text{next}}(u)$ after visiting u and thus no skipped node of the DFS of u will be visited again.

Note that by decreasing $\text{child}_{\text{close}}(u)$ to -1 we actually consume one more token than we had. However, for this last step we only need a constant amount of work which can be accounted for by the processing time of u.

This means that in total we only process $\mathcal{O}(d)$ nodes and do amortized constant work per node as we have claimed.

9.6 Experimental Evaluation

We evaluate our QTM algorithm on the small instances used by Nastos and Gao [NG13], on larger synthetic graphs and large real-world social networks and web graphs. We measured both the number of edits needed and the required running time. For each graph we also report the lower bound b of necessary edits that we obtained using our lower bound algorithm. We implemented the algorithms in C++ using NetworKit [SSM16], our implementation is available online[2]. The implementation of the lower bound uses robin hood hashing[3] [Cel86]. All experiments were performed on an Intel Core i7-2600K CPU with 32GB RAM. We ran all algorithms ten times with ten different random node id permutations.

9.6.1 Comparison with Nastos and Gao's Results

We compare QTM to the algorithm by Nastos and Gao on the five social networks they already considered [NG13], namely karate [Zac77], grass_web [DHC95], lesmis [Knu93], dolphins [Lus+04], and football [GN02] (see Section 2.2 for an introduction). Nastos and Gao [NG13] did not report any running times. We therefore re-implemented their algorithm as described in Section 9.3. Similar to their implementation we use a simple exact bounded search tree (BST) algorithm for the last 10 edits. This bounded search tree algorithm uses none of the optimizations described in Chapter 10. In Table 9.1 we report the minimum and average number of edits over ten runs. Our implementation of their algorithm never needs more edits than they reported[4]. For two of the graphs (dolphins and lesmis) our implementation needs slightly fewer edits due to different tie-breaking rules.

For all but one graph QTM is at least as good as the algorithm of Nastos and Gao in terms of edits. QTM needs only one more edit than Nastos and Gao for the grass_web graph. The QTM algorithm is much faster than their algorithm, it needs at most 2.5 milliseconds while the heuristic of Nastos and Gao needs up to 6 seconds without bounded search tree and almost 17 seconds with bounded search tree. The number of iterations necessary is at most 5. As the last round only checks whether we are finished four iterations would be enough.

9.6.2 Large Graphs

We use a set of seven social networks and four web graphs to examine the scalability and the quality of QTM. Our benchmark set consists of two Facebook graphs [TMP12] (see also Section 2.2.3) and five SNAP graphs [LK14] as social networks (see also Section 2.2.2) and four web graphs from the 10th DIMACS Implementation Challenge [Bad+13; Bol+04; Bol+11; BV04] (see also Section 2.2.1).

[2]`https://github.com/michitux/networkit/tree/upstream/qtm`

[3]`https://github.com/martinus/robin-hood-hashing`

[4]Except on Karate, where they report 20 due to a typo. They also need 21 edits. [Nas15]

Table 9.1: Comparison of QTM and [NG13]. We report n and m, the lower bound b, the number of edits (as minimum, mean and standard deviation), the mean and maximum of number of QTM iterations, and running times in ms.

Name	n	m	b	Algorithm	Edits			Iterations		Time [ms]	
					min	mean	std	mean	max	mean	std
				QTM	72	74.1	1.1	2.7	4.0	0.6	0.1
dolphins	62	159	46	NG w/ BST	73	74.7	0.9	-	-	15 594.0	2 019.0
				NG w/o BST	73	74.8	0.8	-	-	301.3	4.0
				QTM	251	254.3	2.7	3.5	4.0	2.5	0.4
football	115	613	146	NG w/ BST	255	255.0	0.0	-	-	16 623.3	3 640.6
				NG w/o BST	255	255.0	0.0	-	-	6 234.6	37.7
				QTM	35	35.2	0.4	2.0	2.0	0.5	0.1
grass web	86	113	21	NG w/ BST	34	34.6	0.5	-	-	13 020.0	3 909.8
				NG w/o BST	38	38.0	0.0	-	-	184.6	1.2
				QTM	21	21.2	0.4	2.0	2.0	0.4	0.1
karate	34	78	17	NG w/ BST	21	21.0	0.0	-	-	9 676.6	607.4
				NG w/o BST	21	21.0	0.0	-	-	28.1	0.3
				QTM	60	60.5	0.5	3.3	5.0	1.4	0.3
lesmis	77	254	43	NG w/ BST	60	60.8	1.0	-	-	16 919.1	3 487.7
				NG w/o BST	60	77.1	32.4	-	-	625.0	226.4

First, we consider the lower bounds in Table 9.2. Apart from the size of the graphs we report the lower bound and the running time of the lower bound calculation. Both for running times and the size of the bounds we report the average and maximum over ten different node id permutations. The graphs are sorted by the number of edges m. The running times clearly show that the running time does not only depend on m but also on the degrees, i.e. graphs with a lower number of nodes but a comparable number of edges have a higher running time. The average and the maximum do not differ significantly.

It is interesting to see that while for the social networks we are almost always close to the maximum bound of $m/3$, for the web graphs the bounds are much smaller, frequently less than $m/10$.

In Table 9.3, we evaluate two variants of QTM. The first is the standard variant which starts with a non-trivial skeleton obtained by the heuristic described in Section 9.4. The second variant starts with a trivial skeleton where every node is a root. We compare these two variants to determine which part of our algorithm has which influence on the final result. For the standard variant we report the number of edits needed before any node is moved. With a trivial skeleton this number is meaningless and thus we report the number of edits after one round. All other measures are straightforward and are explained in the table's caption.

Even though for some of the graphs the mover needs more than 20 iterations to

Table 9.2: Size and running time of the lower bounds on our benchmark set of social networks and web graphs.

	Name	n	m	Lower Bound		Time [s]	
				mean	max	mean	max
Social Networks	Caltech	769	16 656	5 323.3	5 362	0.0	0.0
	amazon	334 863	925 872	228 062.3	228 231	0.4	0.4
	dblp	317 080	1 049 866	230 119.3	230 354	0.5	0.5
	Penn	41 554	1 362 229	449 398.5	449 570	2.9	2.9
	youtube	1 134 890	2 987 624	883 955.4	886 687	16.5	18.3
	lj	3 997 962	34 681 189	10 703 423.6	10 704 794	82.6	83.1
	orkut	3 072 441	117 185 083	38 757 018.6	38 759 648	917.1	1150.0
Web Graphs	cnr-2000	325 557	2 738 969	228 084.2	229 161	2.8	2.9
	in-2004	1 382 908	13 591 473	817 068.3	818 550	20.4	21.2
	eu-2005	862 664	16 138 468	2 281 512.0	2 285 855	52.6	55.1
	uk-2002	18 520 486	261 787 258	18 633 980.6	18 641 948	409.0	437.7

terminate, the results do not change significantly compared to the results after round 4. In practice we can thus stop after 4 rounds without incurring a significant quality penalty. It is interesting to see that for some of the social networks the initialization algorithm produces a skeleton that induces more than m edits (e.g. in the case of the "Penn" graph) but still the results are always slightly better than with a trivial initial skeleton. This is even true when we do not abort moving after 4 rounds. For the web graphs, the non-trivial initial skeleton does not seem to be useful for some graphs. It is not only that the initial number of edits is much higher than the finally needed number of edits, for two of the four web graphs also the number of edits needed in the end is slightly higher than if a trivial initial skeleton was used.

While the QTM algorithm needs to edit between approximately 50 and 80% of the edges of the social networks, the edits of the web graphs are only between 10 and 25% of the edges. This suggests that quasi-threshold graphs might be a good model for web graphs while for social networks they represent only a core of the graph that is hidden by a lot of noise. For the web graphs, the heuristic never needs more than twice as many edits as the lower bound indicates. As most of the bounds on the social networks are close to the limit of $m/3$, it is not surprising that in particular on the larger graphs they are not as close to the actually needed edits. While our bounds thus confirm that these graphs are far from quasi-threshold graphs, there is still the possibility that quasi-threshold graphs significantly closer than those found by QTM exist.

Concerning the running time one can clearly see that QTM is scalable and suitable for large real-world networks.

Table 9.3: Results for large real-world graphs. Number of nodes n and edges m, the lower bound b and the number of edits are reported in thousands. Column "I" indicates whether we start with a trivial skeleton or not. ● indicates an initial skeleton as described in Section 9.4 and ○ indicates a trivial skeleton. Edits and running time are reported for a maximum number of 0 (respectively 1 for a trivial initial skeleton), 4 and ∞ iterations. For the latter, the number of actually needed iterations is reported as "It". Edits, iterations and running time are the average over the ten runs.

	Name	n [K] / m [K]	b [K]	I	Edits [K] 0/1	Edits [K] 4	Edits [K] ∞	It ∞	Time [s] 0/1	Time [s] 4	Time [s] ∞
Social Networks	Caltech	0.77	5.4	●	15.8	11.6	11.6	8.5	0.0	0.0	0.1
		16.66		○	12.6	11.7	11.6	9.4	0.0	0.0	0.1
	amazon	335	228	●	495	392	392	7.2	0.3	5.5	9.3
		926		○	433	403	403	8.9	1.3	4.9	10.7
	dblp	317	230	●	478	415	415	7.2	0.4	5.8	9.9
		1 050		○	444	424	423	9.0	1.4	5.2	11.5
	Penn	41.6	450	●	1 499	1 129	1 127	14.4	0.6	4.2	13.5
		1 362		○	1 174	1 133	1 129	16.2	1.0	3.7	14.4
	youtube	1 135	887	●	2 169	1 961	1 961	9.8	1.4	31.3	73.6
		2 988		○	2 007	1 983	1 983	10.0	7.1	28.9	72.7
	lj	3 998	10 705	●	32 451	25 607	25 577	18.8	23.5	241.9	1 036.0
		34 681		○	26 794	25 803	25 749	19.9	58.3	225.9	1 101.3
	orkut	3 072	38 760	●	133 086	103 426	103 278	24.2	115.2	866.4	4 601.3
		117 185		○	106 367	103 786	103 507	30.2	187.9	738.4	5 538.5
Web Graphs	cnr-2000	326	229	●	1 028	409	407	11.2	0.8	12.8	33.8
		2 739		○	502	410	409	10.7	3.2	11.8	30.8
	in-2004	1 383	819	●	2 700	1 402	1 401	11.0	7.9	72.4	182.3
		13 591		○	1 909	1 392	1 389	13.5	16.6	65.0	217.6
	eu-2005	863	2 286	●	7 613	3 917	3 906	13.7	6.9	90.7	287.7
		16 139		○	4 690	3 919	3 910	14.5	22.6	85.6	303.5
	uk-2002	18 520	18 642	●	68 969	31 218	31 178	19.1	200.6	1 638.0	6 875.5
		261 787		○	42 193	31 092	31 042	22.3	399.8	1 609.6	8 651.8

9.6.3 Synthetic Graphs

As we cannot show for our real-world networks that the edit distance that we get is close to the optimum we generated synthetic graphs by generating quasi-threshold graphs and applying random edits to these graphs.

We generate each connected component of the quasi-threshold graph as reachability graph of a rooted tree. For generating a tree, we set v_0 as the root node and each node $v_i \in \{v_1, \ldots, v_{n-1}\}$ chooses a parent in $\{v_0, \ldots, v_{i-1}\}$ uniformly at random.

Many real-world networks including social networks exhibit a community size distribution that is similar to a power law distribution [Lan+10]. Therefore, we use a power law distribution $\textsc{Pld}\left([10, 0.2 \cdot n), 1\right)$ for the community sizes. For this, we generate trees of the respective sizes.

For k edits we insert $0.8 \cdot k$ new edges and delete $0.2 \cdot k$ old edges of the quasi-threshold graph chosen uniformly at random. Thus, with these edits applied, the maximum editing distance to the original graph is k. We use more insertions than deletions as preliminary experiments on real-world networks showed that during editing a lot more edges are deleted than inserted.

In Table 9.4, we show the results for the generated graphs. The first column shows the number of random edits we performed. For all generated graphs, our QTM algorithm finds a quasi-threshold graph that is at least as close as the original one. Omitting the initialization gives much worse results for low numbers of edits and slightly worse results for higher numbers of edits. The lower bound is close to the generated and found number of edits for low numbers of edits, for very high numbers of edits it is close to its theoretical maximum, $m/3$.

This shows that the initialization algorithm from Section 9.4 is necessary to achieve good quality on graphs that need only few edits. As it seems to be beneficial for most graphs and not very bad for the rest, we suggest using the initialization algorithm for all graphs.

All in all this shows that the QTM algorithm finds edits that are reasonable but it depends on a good initialization heuristic. An interesting direction for future work would be to make QTM more robust in this regard by implementing strategies to escape such local minima.

9.6.4 Case Study: Caltech

The main application of our work is community detection. While a thorough experimental evaluation of its usefulness in this context is future work we want to give a promising outlook. Figure 9.3 depicts the edited Caltech university Facebook network from [TMP12]. As described in Section 2.2.3, nodes are students and edges are friendships on Facebook. We color the nodes by the dormitories of the students, black indicates that the dormitory is not known. The picture shows that our algorithm succeeds at identifying some of this structure. While some dormitories are clearly separated, in some cases nodes of several dormitories are joined, indicating some kind of overlapping structure.

Table 9.4: Results for the generated graphs

Rand	n	b	I	Edits			It	Time [s]		
Ed.	m			0/1	4	∞	∞	0/1	4	∞
20	100	20	●	34.4	20.0	20.0	2.4	0.0	0.0	0.0
	269		○	36.7	21.4	21.4	3.9	0.0	0.0	0.0
400	100	159	●	420.5	352.0	351.9	3.9	0.0	0.0	0.0
	497		○	372.0	363.5	363.5	3.9	0.0	0.0	0.0
20	1 000	19	●	38.0	19.0	19.0	2.0	0.0	0.0	0.0
	4 030		○	166.4	21.7	21.7	4.1	0.0	0.0	0.0
400	1 000	377	●	585.3	391.2	391.2	3.4	0.0	0.0	0.0
	4 258		○	594.4	393.6	393.6	4.4	0.0	0.0	0.0
8K	1 000	2 876	●	8 268	7 219	7 218.5	5.2	0.0	0.0	0.0
	8 818		○	7 647	7 511	7 490.6	8.3	0.0	0.0	0.0
20	10 000	20	●	47.8	20.0	20.0	2.0	0.0	0.1	0.1
	66 081		○	1 669	69.8	69.8	4.6	0.1	0.2	0.2
400	10 000	390	●	849.3	390.6	390.6	3.2	0.0	0.2	0.2
	66 309		○	2 143	440.7	440.7	4.8	0.1	0.2	0.2
8K	10 000	7 526	●	11 626	7 902	7 902	3.9	0.0	0.2	0.2
	70 869		○	10 184	7 912	7 911	5.1	0.1	0.2	0.3
160K	10 000	53 451	●	157 114	144 885	144 880	5.8	0.0	0.6	0.8
	162 069		○	150 227	148 206	147 892	9.7	0.1	0.5	1.2
20	100 000	20	●	64.3	20.0	20.0	2.0	0.2	1.7	1.7
	833 565		○	17 785	529.9	529.6	5.3	0.8	2.8	3.7
400	100 000	390	●	1 047	391.3	391.3	3.2	0.2	2.6	2.6
	833 793		○	18 319	900.0	899.4	5.5	0.8	2.9	3.9
8K	100 000	7 851	●	18 550	7 889	7 889	3.4	0.2	2.8	2.8
	838 353		○	26 144	8 381	8 381	5.4	0.8	2.8	3.8
160K	100 000	146 268	●	199 558	158 021	158 021	4.6	0.2	3.5	4.1
	929 553		○	193 071	158 031	158 025	6.1	1.0	3.3	4.9
3.2M	100 000	912 228	●	2 728 804	2 647 566	2 647 564	5.8	1.1	12.5	16.8
	2 753 553		○	2 655 538	2 654 738	2 654 736	5.7	3.0	11.9	16.9
20	1 000 000	20	●	68.9	20.0	20.0	2.2	3.6	32.5	32.3
	10 648 647		○	181 540	5 116	5 111	6.0	16.4	54.3	79.7
400	1 000 000	395	●	1 161	395.1	395.1	3.0	3.3	43.8	43.8
	10 648 875		○	182 248	5 523	5 518	6.1	15.9	52.9	78.8
8K	1 000 000	7 906	●	25 085	7 912	7 912	3.5	3.4	50.1	50.0
	10 653 435		○	189 504	13 006	13 001	6.0	16.6	53.4	78.0
160K	1 000 000	157 291	●	369 501	158 808	158 808	4.1	3.5	57.5	58.8
	10 744 635		○	346 462	163 337	163 330	6.4	17.4	54.8	84.5
3.2M	1 000 000	2 793 894	●	3 747 793	3 163 277	3 163 273	5.8	4.7	71.8	101.2
	12 568 635		○	3 820 935	3 164 175	3 163 848	7.5	26.2	78.2	134.7

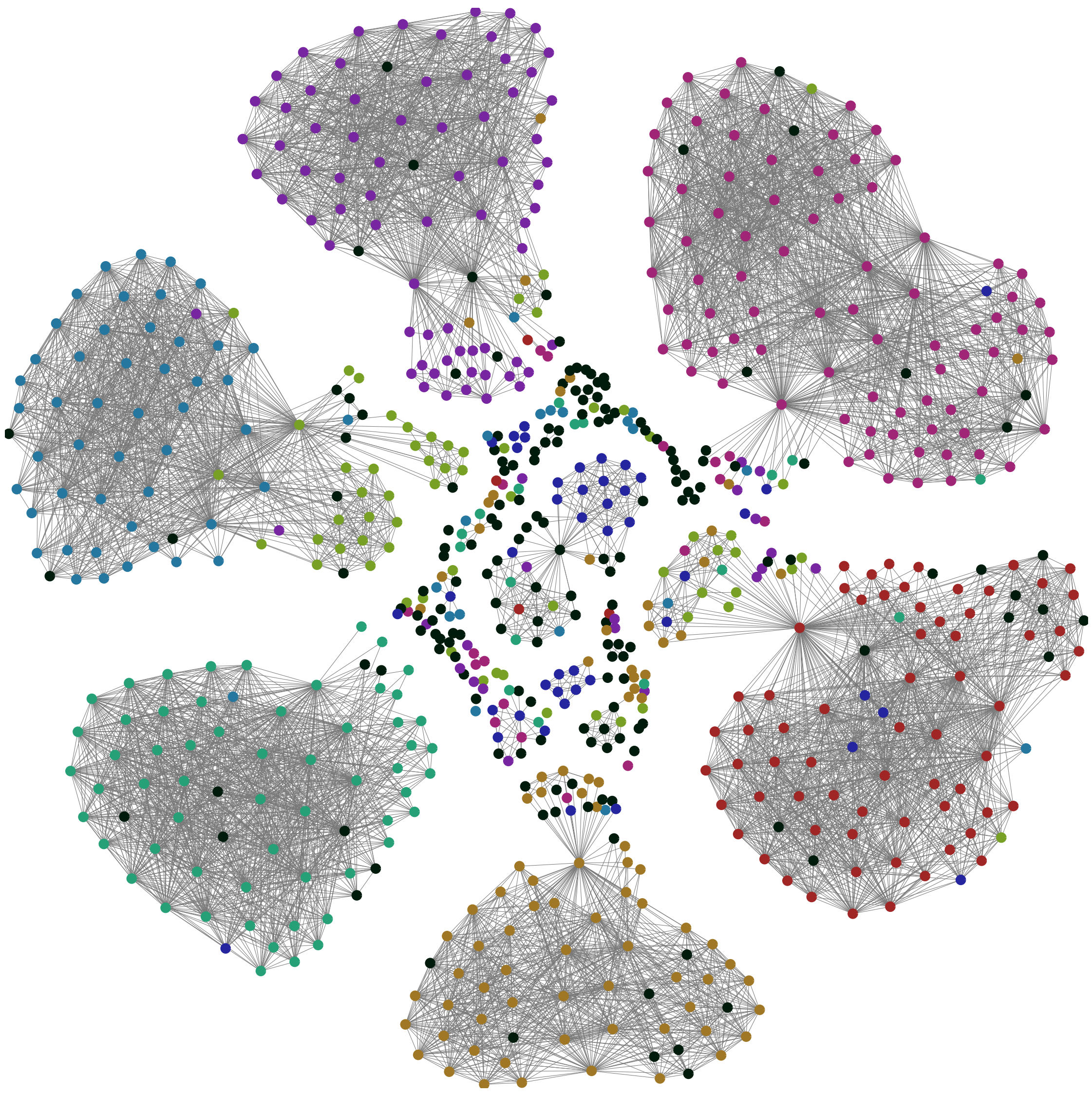


Figure 9.3: Edited Caltech network, nodes colored by dormitories. Black indicates that the dormitory information has not been specified.

9.7 Conclusion

We have introduced Quasi-Threshold Mover (QTM), the first heuristic algorithm to solve the quasi-threshold editing problem in practice for large graphs. As a side result we have presented a simple certifying linear-time algorithm for the quasi-threshold recognition problem. A variant of our recognition algorithm is also used as initialization for the QTM algorithm. In an extensive experimental study with large real world networks we have shown that it scales very well in practice. We generated graphs by applying random edits to quasi-threshold graphs. QTM succeeds on these random graphs and often even finds other quasi-threshold graphs that are closer to the edited graph than the original quasi-threshold graph. A surprising result is that web graphs are much closer to quasi-threshold graphs than social networks, for which quasi-threshold graphs were introduced as community detection method.

Our QTM algorithm can be adapted to consider edit costs per node pair. The actual running time then depends on the actual costs and might be quadratic per round in the worst case. It is the subject of future work to determine if the running time and editing distance of this variant is reasonable in practice. Recently, for the class of P_4-free graphs, finding an inclusion minimal set of edits has been proposed as a fast alternative to exact solutions [Cre20]. As their approach is conceptually similar to QTM, we are investigating if a variant of QTM could be used for inclusion minimal quasi-threshold editing. Further, our QTM algorithm might be adapted for the more restricted problem of threshold editing which is NP-hard as well [Dra+15].

Concerning the running time, we are currently investigating if replacing the priority queue by a bucket priority queue is possible to reduce the running time per round to $\mathcal{O}(n+m)$. Keys in the priority queue are depths in the skeleton forest and might thus be as high as n while we can only spend $\mathcal{O}(\deg(v_m))$ time for moving node v_m, which is why we did not previously consider this option. However, it turns out that for a node v_m, we only need to consider neighbors up to a depth of $2 \cdot \deg(v_m)$. This is because if we want to keep a node u as neighbor, in the edited graph v_m must be connected to all nodes above u in the skeleton forest. If the depth of u is more than $2 \cdot \deg(v_m)$, this incurs more than $\deg(v_m)$ edits as there are more than $\deg(v_m)$ non-v_m-neighbors above u and thus the option to delete all edges to v_m is better.

10 Engineering Exact Quasi-Threshold Editing

This chapter is based on joint work with Lars Gottesbüren, Philipp Schoch, Ben Strasser, Dorothea Wagner and Sven Zühlsdorf [Got+20b; Got+20a]. We thank James Nastos and Mark Ortmann for helpful discussions. Compared to the publication, we have shortened the introduction.

10.1 Introduction

In this chapter we study algorithms to solve editing problems exactly. The *distance* between two graphs G and H, with the same node set, is the minimum number of edge insertions or deletions needed to transform G into H. Given a graph class $\mathcal{C}$ and a graph G, the editing problem asks for a graph $H \in \mathcal{C}$ closest to G.

While the theoretical part of our study considers any $\mathcal{F}$-free edge editing problem for a finite set of subgraphs $\mathcal{F}$, our experimental study considers $\{C_4, P_4\}$-free graphs. Our goal is to improve the running time of exact $\{C_4, P_4\}$-free editing in practice in order to make it feasible at least for small networks. This allows us to study exact solutions of the community detection problem and to verify the quality of heuristics.

10.1.1 Our Contribution

In this chapter, we compare two different methods for solving $\mathcal{F}$-free editing problems. The first is a branch-and-bound FPT algorithm while the second is an ILP. For the FPT algorithm, we propose a novel lower bound algorithm based on local search heuristics for independent sets as well as an improved branching strategy. Additionally, we parallelize our implementation. For the ILP, we engineer several variants of row generation. We assess the running time improvements of the different optimizations for quasi-threshold editing on a large benchmark set of 716 graphs that are connected components of a protein similarity graph. This benchmark set has previously been used to evaluate cluster editing algorithms [Rah+07; Böc+08]. On 75% of the instances, our improved bounds and optimized branching choices yield speedups of one to three orders of magnitude for the FPT algorithm. For the ILP that we solve using Gurobi [Gur20], we are only able to achieve small speedups. With all optimizations, in the median, the FPT algorithm is twice as fast as the ILP, even when enumerating all possible optimal solutions exactly once. Compared with the parallel execution of Gurobi [Gur20], the FPT algorithm achieves better speedups. Additionally, we evaluate an LP relaxation as lower bound. We prove that its bounds are at least as good as our local search bounds. In our experiments, however, it is too slow to be competitive.

Further, we compare our exact solutions with heuristic solutions found by QTM [Bra+15b], see Chapter 9. It turns out that many heuristic solutions are exact and all but one of them are close to the exact solution. Additionally, we are able to solve four out of the five social networks considered in [NG13], of which only one was solved previously [Nas15]. For those social networks, we provide a detailed analysis of the solution space. We show that for none of them there is just a single solution. For one network, we show that quasi-threshold editing gives over 2000 different community structures.

10.1.2 Outline

We start by introducing the preliminaries in Section 10.2. We describe the ILP formulation and the optimizations we apply to it in Section 10.3. In Section 10.4, we then introduce the branch-and-bound FPT algorithm including existing and novel optimizations. In Section 10.5, we present our experimental setup and evaluation. We conclude in Section 10.6.

10.2 Preliminaries

All graphs in this chapter are undirected, unweighted, and finite. Further, no graph has self-loops or multi-edges. A graph $G = (V_G, E_G)$ consists of $n := |V_G|$ nodes and $m := |E_G|$ undirected edges. By $\overline{E_G}$, we denote the complement of the edges. In the following, k denotes the maximum number of edits.

10.3 Integer Linear Programming

In this section, we describe an ILP formulation for $\mathcal{F}$-free editing that is based on an existing formulation for cluster editing [GW89]. Further, we introduce our optimizations based on row generation and modified constraints to make the ILP practical for small instances.

For every node pair $u, v \in \binom{V_G}{2}$ we introduce a variable $x_{uv} \in \{0, 1\}$ which is 1 if the node pair is an edge in the edited graph and 0 otherwise. We add constraints to ensure that no forbidden subgraph $H \in \mathcal{F}$ can be induced in G via an injective node mapping π:

$$\forall H \in \mathcal{F}, \forall \pi \colon V_H \hookrightarrow V_G : \sum_{\{u,v\} \in E_H} (1 - x_{\pi(u)\pi(v)}) + \sum_{\{u,v\} \in \overline{E_H}} x_{\pi(u)\pi(v)} \geq 1 \tag{10.1}$$

The objective minimizes the number of edits:

$$\min \sum_{\{u,v\} \in E_G} (1 - x_{uv}) + \sum_{\{u,v\} \in \overline{E_G}} x_{uv} \tag{10.2}$$

10.3.1 Row Generation

Generating all of the above-mentioned constraints is infeasible, even for small instances. Row generation (also called lazy constraints) aims to speed up ILP solvers by starting with a small subset of the constraints and subsequently adding constraints that are violated in intermediate solutions. We start with constraints for forbidden subgraphs in the input graph. In our experiments, we consider two options to add constraints violated in an intermediate solution: adding either all violated constraints or only one.

The ILP solver uses LP relaxations to prune its search. These can be strengthened by adding constraints from Equation 10.1 that are violated by the LP relaxation. We generate constraints in three steps. First, we consider each node pair $\{u, v\}$ for which the relaxation solution has a value different from the input graph. We edit it, then enumerate the forbidden subgraph embeddings containing u and v, add the constraint that is most violated (i.e., whose left side is furthest below 1) and then revert the edit. Ties are broken uniformly at random. Second, we apply the same procedure to the best heuristic solution found so far. Third, we round the LP solution, i.e., an edge exists iff the corresponding variable is greater than 0.5. We then list forbidden subgraph embeddings in this rounded solution and add the corresponding most violated constraint if there is any. The listing skips forbidden subgraphs for which the corresponding constraint has already been added.

10.3.2 Optimizing Constraints for $\{C_4, P_4\}$-free Editing

If one forbidden subgraph can be transformed into another by a single edit, we can omit a node pair from the constraint for this subgraph. This is similar to the optimization described in Section 10.4.2. For a P_4, this is the node pair consisting of the two degree-one nodes. For a C_4, we can omit any one of its four edges. We always consider all four possibilities, and in the initial constraint generation as well as the basic row generation variant we add all of them. With this optimization, the constraints for C_4s and P_4s are identical.

We can also formulate a constraint for a C_4 that explicitly models that two deletions or one insertion are required:

$$\forall(u_1, u_2, u_3, u_4) \in V_G^4 :\ 0.5 \cdot \sum_{i=1}^{4}(1 - x_{u_i u_{i+1}}) + x_{u_1 u_3} + x_{u_2 u_4} \geq 1 \tag{10.3}$$

10.4 The FPT Branch-and-Bound Algorithm

The FPT algorithm [Cai96] is a branch-and-bound algorithm. For a given maximum number of edits k, it either reports that no solution exists or returns a set of k edits. It works as follows: Find a forbidden subgraph H and branch on all possible edits in H. As H is induced, only edits in H can destroy it and thus one of these edits must be part of the solution. The algorithm is then recursively called for each branch with $k - 1$ remaining edits.

Denote by p the maximum number of nodes in a forbidden subgraph. Finding H can be done trivially in time $O(n^p)$ by enumerating all subgraphs of the required size. For specific sets of forbidden subgraphs, such as $\{C_4, P_4\}$, this can be improved to $O(n+m)$ [Chu08; Bra+15b], see also Section 9.4.

Every pair of nodes in H is a valid edit. The branching factor is therefore $p \cdot (p-1)/2$. The depth of the recursion is bounded by the maximum number of edits k. The total running time is therefore in $O(p^{2k} \cdot n^p)$ for general families of forbidden subgraphs. For quasi-threshold editing the running time is $O(6^k \cdot (n+m))$. This can be improved to $O(5^k \cdot (n+m))$ by applying the optimization described in Section 10.4.2.

For finding the minimum number of edits k_{opt}, the algorithm needs to be executed for increasing values of k until a solution is returned. For a branching factor of 2 or larger, the running time of all $k < k_{\text{opt}}$ together is at most the running time for k_{opt}. Thus, the total running time is dominated by the running time for k_{opt}.

In the following, we describe several optimizations to reduce the number of explored branches in practice. We describe existing techniques for avoiding redundant exploration of branches (Section 10.4.1), for skipping certain branches (Section 10.4.2) as well as lower bounds (Section 10.4.3). We introduce a novel local search lower bound (Section 10.4.4), optimized branching choices (Section 10.4.5), early pruning of branches (Section 10.4.6) and a simple parallelization (Section 10.4.7). In Appendix 10.7 we provide in-depth implementation details.

10.4.1 Avoiding Redundancy

Damaschke [Dam08] proposes to block node pairs to list every solution exactly once. When spawning a search tree node x through editing a node pair, it is neither useful to undo that edit in the sub-search-tree rooted at x, nor is it useful to perform the edit in sibling search trees. While this has been introduced for cluster editing, the technique can be applied to arbitrary $\mathcal{F}$-free editing problems.

Our algorithm maintains a global symmetric $n \times n$-bit matrix. The entries in the matrix correspond to node pairs. If a bit is set, the corresponding node pair must not be edited anymore. We refer to these node pairs as *blocked*. In the following, we describe in detail how these optimizations that were introduced in [Dam08] work.

No Undo Every solution that edits a node pair twice can be improved by not editing the node pair at all. We exploit this observation by setting the bit corresponding to the performed edit when recursing. When ascending from the recursion, we reset the bit. The same optimization has also been used in [Boh15] and is the basis of efficient branching rules for cluster editing [Böc+09].

No Redundancy A possible recursion tree is depicted in Figure 10.1. Note how multiple branches contain the same set of edits, but the edits appear in a different order.

We ensure that every branch enumerates a different set of edits, by unblocking a node pair only after all sibling edits have been explored. In particular, this ensures that every

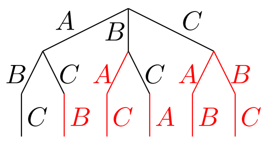


Figure 10.1: Example Recursion Tree. Children are explored from left to right. The "No Redundancy" optimization prunes the red part.

solution is enumerated exactly once. Consider the first recursion level of the example in Figure 10.1. The edits are explored in the following order: A, then B, and finally C. Before descending into the branch of A, we set A's bit. After ascending from A's branch and reverting the corresponding edit, the corresponding bit is not reset. In addition to A's bit, we set B's bit and descend into B's branch. Finally, we ascend from B's branch, leave its bit set, set C's bit and descend into C's branch. After all edits in a recursion level are explored, all bits set in this level are reset, i.e., we reset A's, B's and C's bit.

10.4.2 Skip Forbidden Subgraph Conversion.

Lemma 10.1. *If each forbidden subgraph $A \in \mathcal{F}$ can be transformed into another one $B \in \mathcal{F}$ by one edit, the branching factor of the FPT algorithm can be reduced from $\binom{p}{2}$ to $\binom{p}{2} - 1$.*

There is an edit that transforms a P_4 into a C_4. Clearly, this edit can be skipped. Further, there are four edge deletions that transform a C_4 into a P_4. One of these can be skipped [NG10]. We can choose which one, but as any pair of two edge deletions eliminates the forbidden subgraph, skipping more than one of them might eliminate a necessary branch. Since the branching factor is reduced, this decreases the worst-case running time from $O(6^k \cdot (n+m))$ to $O(5^k \cdot (n+m))$ for quasi-threshold editing.

10.4.3 Existing Lower Bound Approaches

At each branching node, we have a certain number k of edits left. If we can show that the graph needs at least $k+1$ edits, we do not need to explore further branches below that node. Lower bounds have been used for cluster editing [BBK11; HH15] and $\{P_5\}$-free editing [Boh15]. Commonly, they are based on an LP relaxation of the ILP [HH15], or on a disjoint packing argument [Boh15; HH15].

Subgraph Packing. A *node-pair disjoint subgraph packing* P is a set of induced forbidden subgraphs that do not share a node pair. As no edit can eliminate more than one subgraph, $|P|$ is a lower bound on the number of edits required. Taking the previously mentioned optimizations into account, we can include more subgraphs in P by allowing to share blocked node pairs, as they cannot be edited. Further, for each forbidden subgraph a node pair that transforms it into another forbidden subgraph may be shared. In the case of $\mathcal{F} = \{C_4, P_4\}$, the pair of degree-1 nodes of an induced P_4 can be shared. For

C_4, we can choose any edge to share, but it remains the same as long as the C_4 is in the packing.

Finding such a packing can be modeled as an independent set problem [HH15]. The forbidden subgraphs are nodes and every pair of forbidden subgraphs that shares a non-shareable node pair is connected by an edge. A natural greedy heuristic for independent sets is to iteratively add the node that has the smallest degree and then remove all its neighbors from the graph. This can be implemented in linear time by splitting nodes into buckets according to their degree (see e.g. [ARW12]). This heuristic has also been used to calculate lower bounds for cluster editing [HH15]. We are not aware of complexity results of the independent set problem on this special graph class.

In our experiments, we evaluate three bounds based on subgraph packing: 1) A *basic* bound that iteratively adds subgraphs to the packing as they are found. 2) An incremental version of 1) that *updates* the packing as the graph is modified in the branch-and-bound algorithm. After applying an edit, we remove the subgraph that contains the edited node pair. After both editing and blocking, we enumerate and add subgraphs to the bound until it is maximal. 3) A greedy bound based on the *minimum degree* heuristic. In contrast to the first two, this requires storing all forbidden subgraphs. To avoid this in trivial cases, we first apply 2) to the previous bound and only compute a new packing if this fails to prune the branch.

LP relaxation. The optimal solution of the LP relaxation provided in Section 10.3 is an upper bound for the node-pair-disjoint packing problem. This can be shown by considering an LP with just the constraints that correspond to the subgraphs in a packing. Each subgraph in the packing is a node-induced subgraph of G. Therefore, the terms on the left side of its corresponding constraint appear in the objective function exactly as they appear in the constraint, confer Equations 10.1 and 10.2. Each term in the objective function is at least 0, and each group of terms corresponding to a fulfilled constraint sums to at least 1. Since the packing is node-pair disjoint, the constraints do not share any variables and thus groups do not overlap. Therefore, the objective value is at least the number of subgraphs in the packing. Adding more constraints can only increase the objective and thus improve the bound. We can also model blocked node pairs by replacing the corresponding variable by its value. The variables in the constraints are then disjoint again and thus the same argument applies.

10.4.4 Local Search Lower Bound

We propose a lower bound based on a subgraph packing that is computed using an adaptation of the 2-improvements local search heuristic [ARW12] for independent sets. Our local search starts with an initial packing and works in rounds. In each round, it iterates over all forbidden subgraphs in the packing and tries to replace one by two forbidden subgraphs. If this is not possible, it tries to replace one by one. Preliminary experiments have shown that choosing this replacement from those candidates which cover the fewest other forbidden subgraphs leads to significantly higher bounds than considering all candidates. We also found that using this strategy only 70% of the time and otherwise choosing a random replacement is even better. We repeat this procedure until in five consecutive rounds only one-by-one replacements were found. We also terminate the search if the packing remains completely unchanged in a round, or if the packing is large enough to prune the current branch in the search tree. To make this efficient, we approximate the number of forbidden subgraphs that are covered by a certain forbidden subgraph H, by adding up the number of forbidden subgraphs each node pair of H is part of. For the latter we can efficiently maintain counters.

The initial packing is computed with the basic greedy bound. For recursive calls, we update the previous bound as discussed above, before employing local search.

10.4.5 Branch on Most Useful Node Pairs

We can choose any forbidden subgraph for branching on its possible edits, e.g., the first we find. If there is a forbidden subgraph with only one non-blocked node pair, we choose it, as this will lead to just one recursive call. Otherwise, the first node pair we try to edit should ideally lead to a solution, or blocking the edit should prune the search. We propose to prefer forbidden subgraphs whose non-blocked node pairs are part of many other forbidden subgraphs. Then, a single edit can eliminate many forbidden subgraphs (possibly leading to a solution) and blocking the node pairs allows adding many subgraphs to the lower bound. For each forbidden subgraph, we sort its non-blocked node pairs in decreasing order by the number of forbidden subgraphs that contain the respective node pair. The edits of the selected forbidden subgraph are also tried in this order. We select the subgraph to branch on using a lexicographical ordering on these counts. The last node pair is excluded, as there are no branches left to prune. Additionally, if two subgraphs have identical count sequences (up to the length of the shorter one), we prefer the subgraph with the shorter sequence.

10.4.6 Prune Branches Early

Normally, we attempt to prune a branch after applying an edit and descending into recursion. With the optimization from Section 10.4.1, the edited node pair of a recursive call remains blocked after returning from recursion. We update the lower bound to consider this blocked node pair. If the new lower bound already exceeds the remaining number of edits, we can directly prune all subsequent recursive calls, instead of pruning them individually. There are two cases for which we skip the bound update to save running time: If there is only one subsequent recursive call, as we would only prune a single branch, and if the blocked node pair is only part of a single forbidden subgraph, as it cannot yield a better lower bound.

10.4.7 Parallelization

The algorithm can be parallelized by letting different cores explore different branches. Due to our optimizations, not every branch needs the same running time. Therefore, just executing the first branches in parallel is not scalable. Instead, we use a simple work stealing algorithm. Whenever a thread has fully explored its branch, it steals a branch on the highest available level from another thread and further explores it.

10.5 Experimental Evaluation

In Section 10.7 at the end of this chapter we discuss implementation details. The C++ source code[1] of all discussed variants is available online. We use the C++ interface of Gurobi [Gur20] to solve ILPs and LPs. We evaluate our algorithms on a set of 3964 graphs that are connected components of the COG protein similarity data[2] that has already been used for the evaluation of cluster editing algorithms [Rah+07; Böc+08]. The dataset consists of a similarity matrix for each graph. We treat all non-negative scores as edges. Unless stated otherwise, we restrict our evaluation to the 716 graphs that require at least 20 edits. On the 3248 excluded graphs, the maximum running time is less than 0.43 seconds for the FPT algorithm using our local search lower bound. Of these graphs, 1666 require no edits at all. Further, we evaluate our algorithms on a set of 5 small social networks that were already considered by Nastos and Gao [NG13], namely `karate` [Zac77], `grass_web` [DHC95], `lesmis` [Knu93], `dolphins` [Lus+04], and `football` [GN02], see Section 2.2 for an introduction.

All experiments were performed on systems with two 8-core Intel Xeon E5-2670 (Sandy Bridge) processors and 64 GB RAM. We set a global time limit of 1000 seconds. Experiments comparing just FPT variants were executed on 16 different node orders, running 16 node orders in parallel. Due to the memory requirements of Gurobi, this is not feasible for the ILP and the LP bound. For these variants, we run just one instance at a time. For experiments involving ILP variants, we also limit the experiments to 4 node orders, and, for better comparability, we run one instance at a time also for the FPT comparison runs in Figure 10.4. By default, all algorithms terminate at the first found solution, as the ILP is unable to enumerate solutions. Variants with the suffix -All enumerate all solutions. Further, variants with the suffix -MT are parallelized using 16 cores.

10.5.1 Variants of the FPT Algorithm

The baseline branching strategy -F uses the first found forbidden subgraph. Our `Most` branching strategy from Section 10.4.5 is denoted by -M, additional early pruning by -MP. The basic greedy bound is denoted by -G, the incremental update bound by -U, the min-degree heuristic by -MD, our local search lower bound by -LS, and LP relaxations by -LP. The comparison includes the nine variants FPT-G-F-All, FPT-G-MP-All, FPT-U-MP-All, FPT-MD-F-All, FPT-MD-MP-All, FPT-LP-MP-All, FPT-LS-F-All, FPT-LS-M-All and FPT-LS-MP-All.

Figure 10.2 shows how many of the COG dataset instances can be solved within a certain time limit and with a certain number of recursive calls – added over all k's.

[1] `https://github.com/kit-algo/fpt-editing`

[2] `https://bio.informatik.uni-jena.de/data/#cluster_editing_data`

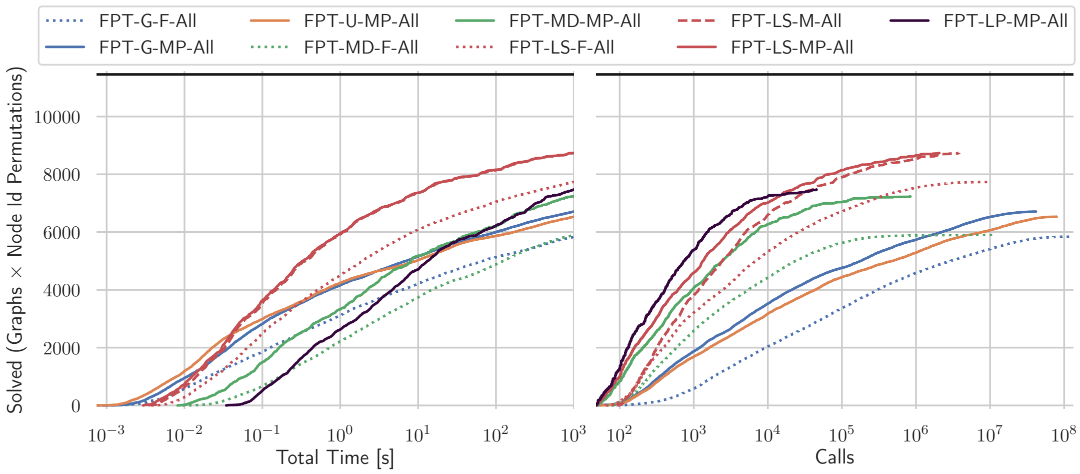


Figure 10.2: Number of permutations of graphs of the COG dataset that require at least 20 edits and can be solved within a certain total running time / with a certain number of recursive calls (and extra lower bound updates for -MP). The horizontal black line indicates the total number of graphs and node permutations that require 20 or more edits, including unsolved instances.

Additional lower bound calls due to -MP count extra. An instance is a single node id permutation of a graph, i.e., every graph is counted 16 times. Of the 716 graphs we are able to solve 547 within the 1000 second time limit. Below, we also compare calls and running times per instances.

For comparing branching strategies, we fix the local search algorithm -LS as the lower bound. The median factor of additional calls needed by -M over -MP is 1.9 and by -F over -MP is 3.36, restricted to instances solved by both algorithms. While the median speedup of -MP over -F is 3.11, it is just 1.06 over -M. On 5% of the instances, the speedup is at least 56.62 and 1.24, respectively. This shows that for -M the improvement in the number of calls directly leads to similar running time improvements, while early pruning just reduces calls.

For comparing lower bound algorithms, we fix -MP as the branching strategy. There is an inherent trade-off between the number of recursive calls and the time spent per call, with a sweet spot that gives the best overall running time. The basic greedy bounds need 10 to 24 times as many calls as the other bounds in the median. The recomputed greedy bound -G is slightly better than the updated one -U, -LP is the best, followed by -LS and -MD.

Nonetheless, for very small time limits, -U solves the highest number of instances. For larger time limits, reducing the number of calls pays off, though not at any cost. The median speedup of min-degree over the LP is 2.16 while needing 47% more calls in the median. Local search avoids their substantial memory overhead and spends significantly less time per call than both. It needs 83% of the calls of -MD while being a factor of

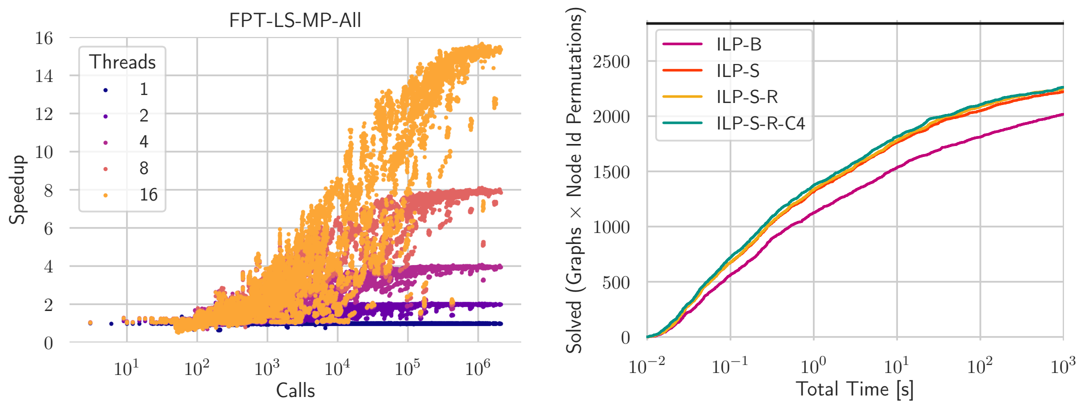


Figure 10.3: Speedup of FPT-LS-M-All-MT and comparison of the different ILP variants on 16 (left) and 4 (right) node id permutations of the 716 COG graphs that require at least 20 edits.

12.36 faster in the median. It is never slower, and on 5% of the instances even more than 137 times faster than -MD.

Comparing the state-of-the-art FPT-MD-F-All algorithm to our FPT-LS-MP-All algorithm, we need 4.33 times less calls and are 46.06 times faster in the median. We are never slower, on 75% of the instances more than 16 times and on 5% of the instances more than 1044 times faster. In conclusion, our local search lower bound gives high-quality bounds while being fast. Our branching rules reduce the running time by another small factor while early pruning mainly reduces the number of calls. Overall, we achieve a speedup of one to three orders of magnitude over the state-of-the-art.

10.5.2 Parallelization

The left part of Figure 10.3 reports the speedup of FPT-LS-MP-All-MT over its sequential counterpart FPT-LS-MP-All, the sequentially fastest variant on the COG dataset. We show the speedup with 1, 2, 4, 8 and 16 cores in comparison to the number of recursive calls and lower bound calculations. For each graph and permutation, we plot the speedup on the last value of k for which the sequential version of the algorithm terminated within the time limit. With only few recursive calls we cannot expect a good speedup. For a high number of recursive calls, FPT-LS-MP-All-MT achieves almost perfect speedup for all numbers of cores on many graphs. As the algorithm is executed with increasing values of k, for some graphs only the last value of k needs a high number of calls and thus the overall speedup is not perfect even though in sum the number of calls is high.

10.5.3 Variants of the ILP

Figure 10.3 (right) shows the impact of the different optimizations on the ILP, when enabled one after another. We denote adding only one violated constraint by -S, adding

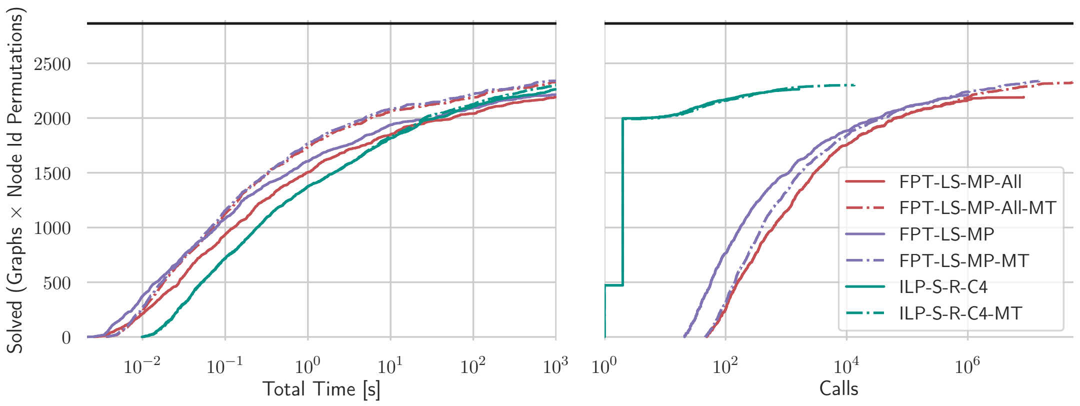


Figure 10.4: Comparison of the ILP to the FPT algorithm on 4 node id permutations of the 716 COG graphs that require at least 20 edits.

constraints during relaxations by -R, and specialized C_4 constraints by -C4. The baseline is ILP-B, where row generation always adds all violated constraints for intermediate solutions.

In the median, ILP-S is just 5% faster than ILP-B. While on 95% of the instances it is at most 20% slower, it is more than 44 times faster on 5% of them, which explains the gap in Figure 10.3. ILP-S-R is not faster in the median, but on 95% of the instances at most 12% slower and on 5% it is at least 73% faster. The C_4 constraints make the ILP 12% faster in the median, at most 26% slower on 95% and at least 95% faster on 5% of the instances. With all optimizations, the ILP solves 568 graphs. We also tried providing a heuristic solution from QTM [Bra+15b] to Gurobi, but the improvement was even smaller and disappeared in parallel.

Figure 10.4 compares the best ILP and FPT algorithms with and without -MT in terms of running time and recursive calls. For the FPT algorithm, stopping at the first solution is not slower on 95%, more than 52% faster on 50% and more than 3 times faster on 5% of the instances. Multi-threading incurs a measurable overhead. Compared to FPT-LS-MP-All, FPT-LS-MP-All-MT is at most 16% slower on 95%, 78% faster in the median and more than 12 times faster on 5% of the instances. When stopping at the first solution, this decreases to 24% slower, 1% faster and 10 times faster, as more branches that do not lead to a solution are explored in multi-threaded mode. FPT-LS-MP-MT is still 4% faster than FPT-LS-MP-All-MT in the median, at most 3% slower on 95% and at least 68% faster on 5% of the instances.

The parallel ILP is at most 5% slower on 95%, as fast in the median and more than 52% faster on 5% of the instances than the sequential ILP. Thus, the parallelization helps the FPT algorithm more than the ILP. A likely cause is that Gurobi needs much less search nodes than the FPT algorithm which offer less potential for parallelism – on 50% of the instances at least 185 times less, and on many graphs even just one or two, see Figure 10.4.

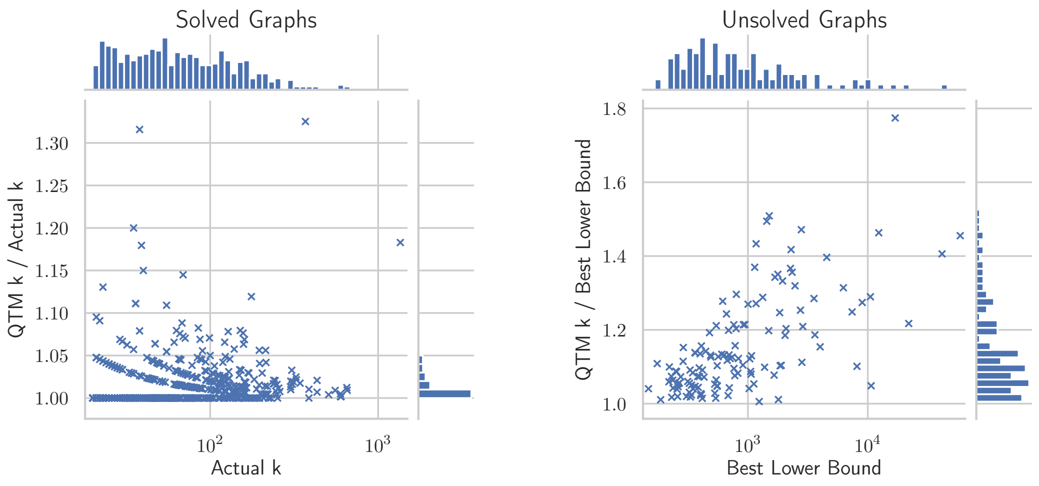


Figure 10.5: Comparison of heuristic solutions of QTM and the exact number of edits k for solved graphs (left) or the best lower bound for unsolved graphs (right) achieved by the FPT algorithm or the ILP. For readability, we exclude one solved graph at $k = 64$, where QTM needed 202 edits.

The speedup of FPT-LS-MP over ILP-S-R-C4 is at least 0.59 on 95%, 3.25 in the median and at least 10.72 on 5% of the instances. For FPT-LS-MP-All, this decreases to 0.29, 2.10 and 7.02. In parallel, the speedups are 1.09, 3.41 and 16.45 for all solutions, and 1.34, 3.67 and 18.14 for the first solution. Single-threaded, the ILP solves more instances within 1000 seconds than the FPT algorithm, indicating that for difficult instances better bounds are more important. Overall, the FPT algorithm is often faster than the ILP, in particular in parallel and even when listing all solutions.

10.5.4 Comparison to QTM

Figure 10.5 compares the results of our heuristic Quasi-Threshold Mover (QTM) [Bra+15b] (see Section 9.5) with exact results for solved and the best lower bounds for unsolved graphs. We use the maximum value of k achieved for any permutation by FPT-LS-MP-MT and by ILP-S-R-C4 with and without -MT. If any of the algorithms solved the graph, we list it in the left part, otherwise in the right part. For QTM, we report the minimum k that QTM found over 16 runs. Again, the plot excludes 3248 graphs that require less than 20 edits. Of those, QTM solved 3172 exactly, 56 with offset 1, 15 with offset 2 and 5 with offset 3. Of the remaining graphs, 588 are solved and 128 are unsolved. Of the solved graphs, QTM solved 319 graphs exactly. For none of the unsolved graphs, QTM matches the lower bound. Apart from one outlier, QTM never needs more than 1.4 times the number of edits of the exact solution and never more than 1.8 times the number of edits indicated by the best lower bound. For 95% of the 716 graphs, QTM needs at most 1.22 times the edits of the exact solution or the lower bound.

Table 10.1: Overview of the social network graphs. Using the algorithms FPT-LS-MP and ILP-S-R-C4 with 1 and 16 cores, we report the maximum k that finished within 1000 seconds, and the minimum time (T) over all permutations that is needed to find the first solution. In the case of `football`, we report the time needed to show that there is no solution with that k.

			FPT				ILP			
			1 core		16 cores		1 core		16 cores	
Graph	n	m	k	T [s]	k	T [s]	k	T [s]	k	T [s]
karate	34	78	21	0.01	21	0.01	21	0.02	21	0.03
lesmis	77	254	60	0.17	60	0.13	60	0.96	60	0.97
grass_web	75	113	34	1.81	34	0.21	34	2.91	34	2.83
dolphins	62	159	70	126.54	70	18.57	70	23.81	70	12.10
football	115	613	223	929.55	228	649.94	235	1000.01	237	1000.05

10.5.5 Social Network Instances

Table 10.1 shows an overview of the social networks with results for FPT-LS-MP and ILP-S-R-C4. Both solve `karate` and `lesmis` in less than a second, and `grass_web` within 3 seconds, with the FPT algorithm being faster. Even though `lesmis` is both larger than `grass_web` and requires 60 edits instead of 34, both algorithms are significantly slower on `grass_web`. This shows that their performance depends on the specific structure of the graph and not just the graph size and k. For `dolphins`, the ILP is faster than the FPT algorithm. For all graphs, the FPT algorithm scales better with the number of cores. None of the algorithms can solve the `football` network. We show that there is no solution for $k \leq 223$, $k \leq 228$ using the FPT algorithm with 1 or 16 cores respectively, and $k \leq 235$, $k \leq 237$ using the ILP with 1 or 16 cores respectively. The previously best known upper bound was 251, computed with QTM [Bra+15b] in 2.5ms, see also Section 9.6.1. In 1000 seconds, the ILP shows a new upper bound of 250. For the smallest three social networks, we verify that the best heuristic solutions Section 9.6.1 [Bra+15b] are exact. QTM needs 72 edits on `dolphins`, whereas 70 edits are optimal.

10.5.6 Evaluation of the Found Solutions

The FPT algorithm offers the possibility to list all solutions exactly once. In this section, we show the number of solutions found for the COG dataset. Further, we examine the solutions that are found on the four solved social network instances in detail, comparing them concerning the community detection application of quasi-threshold editing.

Figure 10.6 shows the number of found solutions for all solved graphs of the COG dataset. We plot the number of solutions for the graphs grouped by k to see if there is any correlation between k and the number of solutions. For some graphs with $k > 100$, there are over a million solutions found. However, there seems to be no strong correlation

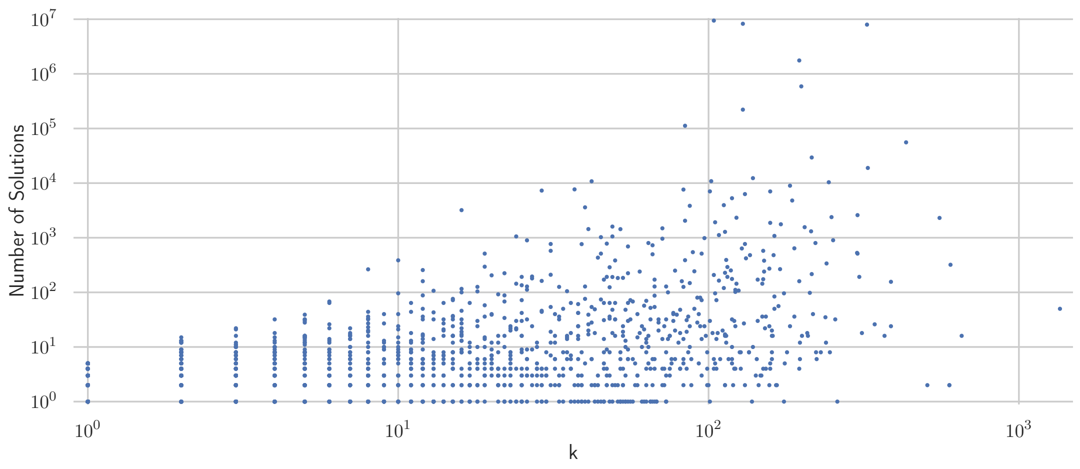


Figure 10.6: Number of solutions for all solved graphs of the COG dataset sorted by k with $k > 0$.

between k and the number of solutions found. Nevertheless, this shows that in many cases there is not one clear solution for a graph.

To examine how this affects the graph clustering application suggested by [NG13], we take a closer look at the solutions for the four solved social networks using NetworKit [SSM16]. Table 10.2 summarizes the found solutions. The number of found solutions ranges from 24 on `dolphins` up to 3006 for `grass_web`. Nastos and Gao [NG13] propose to use the connected components of the edited graphs as clusters. While two closest quasi-threshold graphs might use different edits, they can still induce the same clustering as some edits only affect edges inside clusters. Therefore, we also examine the number of different clusterings found. For `karate`, the 896 solutions induce only 12 different clusterings, while for `grass_web`, there are 2250 different clusterings induced by 3006 solutions. This shows that there can be quite some variance in terms of the found clusterings even between exact solutions. The number of clusters remains rather stable, on the other hand. Figure 10.7 shows two solutions of `grass_web` with two different clusterings. The brown cluster in the right solution is split into the brown, bright-green and dark-blue clusters in the left solution. Further, two nodes that are marked by a blue circle in the figures switch their cluster assignment.

To see if there is something all solutions can agree on, we examine the edits that are common to all solutions. Out of the 21 edits necessary for `karate`, there are 11 common edge deletions. These induce a cut into two clusters, which is also the cut found in [Zac77]. For `grass_web`, there are 1 edge insertion and 11 edge deletions common to all solutions which also only split the graph into two parts – compared with the necessary 34 edits and the up to 14 clusters in each solution. For `lesmis`, there are 4 edge insertions and 45 edge deletions common to all solution which induce 6 clusters – this shows a structure that is a lot more stable. On `dolphins`, there are 5 edge insertions and 56 edge deletions common

Table 10.2: Summary of the solutions found. For each graph, we report the number of different solutions, the number of different induced clusterings, the minimum and maximum number of clusters in the different solutions, the number of insertions and deletions common to all solutions, the number of clusters obtained when just applying the common edits, the total number of different insertions and deletions and the number of clusters obtained when intersecting all found clusterings.

		#Solu-	#Clus-	#Clusters		Common			Union		
Graph	k	tions	terings	Min	Max	Ins.	Del.	Cl.	Ins.	Del.	Cl.
karate	21	896	12	2	4	0	11	2	13	27	7
grass_web	34	3006	2250	11	14	1	11	2	11	45	22
lesmis	60	384	192	8	12	4	45	6	10	63	16
dolphins	70	24	8	12	13	5	56	9	11	71	16

to all solutions, i.e., each solution only adds 9 further edits. These common edits already induce 9 clusters which is close to the 12 or 13 clusters found in the individual solutions.

Additionally, we look at the number of edits in the union of solutions. For all graphs there are more edge deletions than insertions. Even if all edge insertions were in a single solution, in all graphs but `karate` there were more than two times more edge deletions than insertions. Further, we calculate the intersection of all found clusterings to obtain the largest clusters that are not split in any solution. For `karate`, this gives us 7 clusters that split both of the two parts into further parts. For `grass_web`, we even obtain 22 clusters (compared with at maximum 14 clusters in an individual solution). For `lesmis` and `dolphins`, we obtain 16 clusters, i.e., a value relatively close to the up to 12 or 13 clusters that are found in individual solutions.

This analysis shows the power of being able to enumerate all solutions. We can not only determine how stable the clustering structure of a graph is, we can also obtain the smallest components on which all different solutions agree – or large clusters, where all solutions agree that they should be split. This could also be used to obtain overlapping clusters by assigning nodes that frequently change between clusters to several clusters.

10.6 Conclusion

We have introduced optimizations for two different approaches to solving any $\mathcal{F}$-free edge editing problem. We evaluate our optimizations for the special case of quasi-threshold editing on a set of 716 protein interaction graphs. For the first approach, the FPT algorithm, we show that the combination of good lower bounds with careful selection of branches allows to reduce the running time by one to three orders of magnitude for 75% of the instances. For the second approach, an ILP, we evaluate several variants of row generation and show that they achieve small speedups. We show that the FPT algorithm is slightly faster than the ILP, with a larger margin in parallel, and it can easily

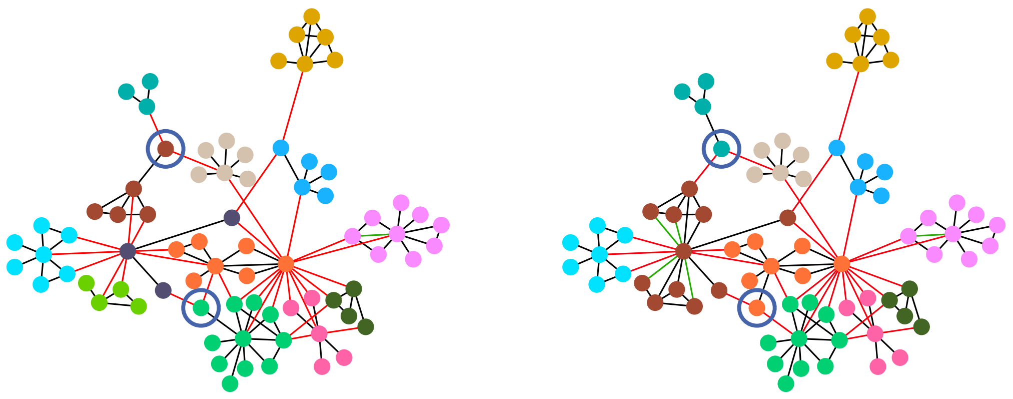


Figure 10.7: Two solutions of `grass_web`. Red edges have been deleted, green edges have been inserted. Nodes are colored by connected component in the edited graph. The two blue circles denote two nodes that changed clusters.

enumerate all optimal solutions. For our heuristic editing algorithm QTM, we show that on 95% of the instances, it needs at most 22% more edits than our exact solutions or lower bounds indicate.

Comparing the structure of exact vs. heuristic solutions might give further insights how to improve heuristics. Exact FPT algorithms could be further improved by better bounds, possibly based on LP relaxations. As the COG benchmark set actually contains edit costs, an extension of our optimizations to the weighted editing problem could be investigated.

10.7 Implementation Details

In this section, we document various details of our implementation. We first describe our graph data structure and how we iterate over it. In Section 10.7.1, we describe how we list forbidden subgraphs. We maintain subgraph counters that we describe in Section 10.7.2. In Section 10.7.3, we describe for each lower bound algorithm how it is implemented using the aforementioned subgraph listing algorithms. In Section 10.7.4, we describe the implementation of our branching strategy. Finally, in Section 10.7.5, we describe our parallelization in detail.

We store our graph as an adjacency matrix with 1 bit per node pair. To enumerate edges or neighbors, we use special CPU instructions to count leading zeros in a copy of a 64 bit block of this matrix. We then remove the found 1-bit from the 64 bit block and count again. If the current 64 bit block contains only zeros, we move to the next one. All but the largest 24 graphs in our benchmark set have at most 320 nodes, thus requiring at most five 64 bit blocks per row of the matrix. The largest graph has 8836 nodes and thus requires 139 64 bit blocks per row, but its average degree is also 64, therefore on average almost every second block contains a 1-bit. Thus, for almost all of the graphs we consider, bit matrices seem an appropriate choice in terms of memory usage and enumeration efficiency. For the larger graphs, adjacency arrays might be a better choice but as we are far from solving them, we did not further explore this. Adjacency matrices have the advantage that we can easily combine multiple rows to list common neighbors or exclude neighbors of another node, a feature that we use for subgraph listing as described in the following. Let $A[i, j]$ denote the entry of the adjacency matrix in row i and column j. We use matrix slice notation $A[:, j]$ to denote column j, and $A[i, :]$ to denote row i, i.e., the neighbors of node i.

10.7.1 Subgraph Listing

To select subgraphs for branching and calculating lower bounds, we need to enumerate all forbidden subgraphs. In preliminary experiments, we found that enumerating forbidden subgraphs on demand does not only require much less memory than storing them, but is also much faster. Our implementation provides two methods for this: a global one that lists all forbidden subgraphs and a local one that lists all subgraphs containing a certain node pair. The latter is required to efficiently implement our local search lower bound and the branching on most useful node pairs. For simplicity, our descriptions focus on $\{C_4, P_4\}$-listing but our source code works for arbitrary $\{P_l, C_l\}$, $l \geq 4$ and $\{P_l\}$, $l \geq 2$.

Global Listing. For the listing of all forbidden subgraphs, we enumerate all edges. We consider each edge $\{u_2, u_3\}$ as the central edge and then enumerate edges $\{u_1, u_2\}$ in the outer loop, and $\{u_3, u_4\}$ in the inner loop, to complete the P_4 or C_4. For listing candidates u_1, we directly exclude neighbors of u_3 by only iterating over $A[u_1, :] \wedge (\neg A[u_3, :])$. We list candidates u_4 analogously. Fixing the central edge ensures each induced P_4 is listed exactly once. We list each C_4 four times, which we use for trying different shareable node pairs for the packing lower bounds, as each edge deletion transforms the C_4 into a P_4.

Local Listing. For the listing of forbidden subgraphs that contain a certain node pair $\{u, v\}$, we need to consider all positions of $\{u, v\}$ in the forbidden subgraph. If $\{u, v\}$ is an edge, this means that apart from the case where $\{u, v\}$ is the central edge, we also need to consider the case where we extend the path twice on each side. If $\{u, v\}$ is not an edge, the case where $\{u, v\}$ consists of the two degree-1-nodes of the P_4 can be omitted due to the optimizations discussed in Section 10.4.2. We only need to find common neighbors $x \in A[u, :] \wedge A[v, :]$ of u and v. These are part of the central edge. We try extending the path by one edge from u and v separately, i.e., iterate over $A[u, :] \wedge (\neg A[x, :])$ and $A[v, :] \wedge (\neg A[x, :])$.

Listing For Lower Bounds. For lower bounds, we are only interested in forbidden subgraphs that do not contain any node pair that is already used in the lower bound. We maintain a bit matrix L where all node pairs that are already used in the bound are set to 1. By using $A[u, :] \wedge (\neg L[u, :])$ instead of $A[u, :]$ for neighbors of u and $(\neg A[u, :]) \wedge (\neg L[u, :])$ for non-neighbors, we can directly exclude these node pairs from listing.

Excluding Specific Node Pairs From Listing. To branch on its node pairs or to check if a subgraph can be added to a lower bound, we need to enumerate its node pairs. For this, we implicitly exclude blocked node pairs as well as $\{u_1, u_4\}$, which is the node pair of degree one in a P_4. As mentioned before, we always list a C_4 four times, and thus omit a different edge $\{u_1, u_4\}$ in each enumeration. This lets the lower bound algorithms select the best node pair to share or the branching strategy select the best node pair to exclude.

10.7.2 Subgraph Counters

In our `Most` and `Most Pruned` branching strategies and the local search lower bound, we want to select the subgraph whose node pairs cover the most or least other forbidden subgraphs. For this, we maintain a counter for each node pair in how many forbidden subgraphs it is contained. Whenever a node pair is edited or blocked/unblocked we update the counters. When blocking a node pair, we store its previous counter on a stack so that it can be easily restored when unblocking, and set the current counter to zero. Note that our counters count a C_4 three times for edges and four times for non-edges due to listing the C_4 four times and omitting one of the edges each time. We also maintain the sum of the subgraph counters to be able to quickly check if there are any forbidden subgraphs at all.

10.7.3 Lower Bound Algorithms

Each of our lower bound algorithms has both a thread-state that is maintained once per thread and a call-state that is copied for every recursive call. We compute an initial lower bound on the input graph and start our search for k_{opt} from this bound instead of 0. The call-state is initialized once during this initial lower bound calculation and then used as initialization for all ks that we try. For most algorithms, the call-state contains the previously calculated lower bound as an array of subgraphs (node tuples) in the packing. We pass the call-state down into recursive calls, but not back up. The rationale behind this is that we need to remove at most one forbidden subgraph from the bound when descending into recursion, whereas no longer blocked node pairs would force a lot of subgraphs to be removed when returning from a recursive call.

Basic Bound. For the basic bound, we globally enumerate forbidden subgraphs H and add H to the packing P if none of its node pairs are used by another graph in the packing. This is done in each recursive call with an initially empty packing. After the one pass, P is inclusion-maximal. We maintain a bit matrix C for node pairs covered by the packing in the thread-state. When adding H to P, we mark its node pairs in C. We also supply a reference of C to the listing algorithm to skip subgraphs we cannot use. As additional bits in C are set during the listing, it does not skip all subgraphs we cannot use. For example, the listing does not check the central node pair again. In preliminary experiments this still gave a small speedup.

Updates. For the basic bound with updates we pass the packing P through the search tree. Before descending into recursion, we remove the subgraph H that contains the edited node pair $\{u, v\}$ from P, if it exists. If possible, we add subgraphs H' to P that share a node pair with H or contain $\{u, v\}$, to make P inclusion-maximal. This is done using the local listing. Similar to before, the local listing skips some subgraphs touched by C. We store a list of subgraphs in the packing in the call-state. For memory efficiency, we do not store C in the call-state but instead recompute it from scratch when modifying the bound.

Local Search. The local search lower bound also just maintains the list of subgraphs used in the bound in its call-state. The initial update works as described above. To find candidates for replacing one subgraph by one or more subgraphs, we use the local listing. In each round, we try to replace each subgraph H in the packing once. For this, we first remove H from P and then use the local listing on all node pairs of H to obtain the set R of subgraphs that could replace H. From R, we obtain candidates that can be inserted together. For each subgraph $H' \in R$ we first insert it into P, and then iterate over the rest of R, trying to insert. If at least one additional candidate was found, we keep them in the packing. Otherwise, we can only replace H by H'. With 70% probability we take the H' that covers the fewest other forbidden subgraphs, and with 30% probability we choose a random one from R.

We apply several optimizations to speed up the search for additional candidates. For each node pair of H, we store a separate list of candidates. This allows us to skip candidates that use a node pair that is used by an already included candidate, without considering each candidate separately. To avoid trying the same candidate twice, we also list candidates only for the first node pair they contain by excluding the already considered node pairs from the candidate search for subsequent node pairs.

Min-Degree Heuristic. The min-degree heuristic is based on the independent set formulation where a subgraph is a node and two nodes are connected by an edge if the corresponding subgraphs share a node pair. A good lower bound then corresponds to a large independent set. For independent sets, the min-degree heuristic iteratively adds the node with the smallest remaining degree to the independent set and then deletes it and its neighbors from the graph. Instead of explicitly constructing this graph model, we translate this formulation back to forbidden subgraphs.

We iteratively add subgraphs to the bound whose node pairs are shared with the least number of subgraphs that can still be added to the bound. To implement it, we need to explicitly maintain the "degree" of every subgraph in a priority queue as it is changing over time as more and more subgraphs are added to the bound. For this, we (temporarily) store an explicit list of all forbidden subgraphs that we obtain through global listing. Further, for every node pair we store a list of subgraphs it is part of by storing their indices in the list of subgraphs. Similarly, we store these list indices in the priority queue. This allows to efficiently identify the elements that need to be updated or removed from the priority queue. For a subgraph H, we initially use the sum over all node pairs of

H of the number of subgraphs that contain the node pair as key. This might count the same subgraph several times, but is more efficient to calculate. Preliminary experiments showed that this is faster than calculating the actual number of subgraphs with whom H shares a node pair.

Whenever we take a subgraph H from the priority queue and add it to the bound, we need to remove its neighbors from the priority queue and update their neighbors' degrees accordingly. To obtain the neighbors N of H, we iterate over H's node pairs and list all subgraphs they are part of. We remove each subgraph $n \in N$ from the priority queue and for each of n's neighbors, we decrement its key in the priority queue by one.

For the priority queue, we use a bucket priority queue. As nodes only need to be moved between adjacent buckets, we can maintain all buckets in one large array and move elements between buckets by swapping them to the boundary and then adjusting the boundary.

LP Relaxation. The LP for our *LP bound* corresponds exactly to the ILP formulation shown in Section 10.3 with the optimization of omitting one node pair as described in Section 10.3.2. Our main goal for the implementation of the LP bound was to have a comparison with a lower bound algorithm that is guaranteed to prune at least as good as the packing-based lower bounds. For this reason, we ensure that the LP always contains the constraints that correspond to forbidden subgraphs that could also be used in the packing-based lower bound. Similar to the ILP, we initialize the LP with all constraints that correspond to forbidden subgraphs in the input graph. Whenever we edit or block a node pair, we fix the value of its corresponding value to 1 or 0, depending on whether it is now connected by an edge or not. When a node pair is edited, it is always blocked and thus all constraints that correspond to forbidden subgraphs that no longer exist in the edited graph are trivially fulfilled and there is thus no need to remove them explicitly. After each edit, we add constraints that correspond to forbidden subgraphs that contain the edited node pair. After undoing an edit, we remove them again, as the LP solver slows down when the LP contains a lot of constraints.

10.7.4 Branching Strategies

As described in Section 10.4.5, we want to prefer subgraphs that contain at most one non-blocked node pair, as we know this edit has to be applied. In a call-state like those of the lower bound algorithms, we store both this list of subgraphs, and a flag indicating whether the branch can be pruned. Such subgraph can only appear when blocking or editing a node pair $\{u, v\}$. Every time this happens, we enumerate all subgraphs H containing $\{u, v\}$. If H contains exactly one non-blocked node pair, we store it. If H contains only blocked node pairs, the current branch can be pruned immediately because H cannot be destroyed. Before the branching strategy selects a forbidden subgraph, we first check if the flag is set and return an empty list of node pairs, if so. Otherwise, we iterate over the list of subgraphs in the call-state and return the only non-blocked node pair of the first subgraph of the list where this node pair has not been edited yet. Note that two of these subgraphs might contain the same non-blocked node pair, thus after the first of them has been selected, the second becomes invalid.

Only if the flag is not set and there is no subgraph with exactly one non-blocked node pair, we apply the actual branching strategy. In our `Most` and `Most Pruned` branching strategies, we avoid listing all forbidden subgraphs. Instead, we first identify those node pairs that are part of the maximum number of subgraphs using the subgraph counters introduced in Section 10.7.2. Due to our lexicographical ordering, we are only interested in those subgraphs that contain these node pairs. We use our local listing to enumerate them and select the maximum as described in Section 10.4.5. The output of the branching strategy is a sorted list of node pairs and a flag whether the graph is solved. We set this flag if the sum of the subgraph counters is zero.

10.7.5 Parallelization

In our parallelization, different threads explore different branches of the search tree. We maintain a global queue of work packages that represent roots of unexplored branches. To achieve a scalable parallelization we want to generate few work packages that have a lot of recursive calls left. Due to our optimizations, we cannot know in advance how many calls are left for a certain branch. Even branches that start at the root of our recursion tree might be pruned after a single call. Therefore, we need to generate work packages as we explore the search tree, i.e., employ work-stealing.

Each work package contains the number of remaining edits, the graph, the blocked node pairs, the subgraph counters and the call-states of the lower bound and the branching strategy (can be empty). Hence, creating a work package for every call is too expensive, as we would need to copy these data structures, whose memory consumption is quadratic in the number of nodes. Passing them through the search tree, and updating on-the-fly is fast, but a work package constitutes the root of a new search tree and thus requires a copy. Therefore, we only create work packages when the global work queue contains less work packages than the number of threads.

When a worker finishes one recursion tree, it takes another work package from the queue. If there is none left, it waits until either work becomes available or the algorithm is finished. The latter is indicated either by the fact that no thread has a work package anymore (we keep a counter how many threads are currently working), or a global flag that is set when the first solution has been found and not all solutions shall be listed. This flag is also checked in each recursive call to ensure that if one thread finds a solution, all other threads terminate.

At the beginning of every recursive call, we check if work packages shall be generated. A simple approach would be to split the recursive calls of the current search tree node into work packages. Unfortunately, this does not scale well, as we would predominantly create work packages on deeper recursion levels where only few edits remain. Instead, we split off unexplored branches from the top of the current recursion tree, where we hope the most work is left. For this, we explicitly maintain the current recursion path of each worker thread.

Each element of the path contains the node pairs to branch on, the call-states of the bound and the branching strategy for each of these branches and an index that indicates the next branch to be explored. After potentially generating work packages, we invoke the lower bound calculation. We then check if the recursive call can be pruned because of the bound, because there are no more edits left or because a solution has been found. If not, we create its element in the path. For this, we obtain the node pairs for the next recursion level from the branching strategy. We then create copies of the call-states for all of them and update them such that each of them can be directly used to create a work package. This ensures that work package generation, which happens inside a global lock, is quick and does not need to update call-states. For the early pruning (see Section 10.4.6), we also directly check if calls can be pruned and if yes, we directly remove them from the node pairs.

For the actual recursive calls, we iterate over the node pairs in the element of the path. We advance the index that indicates the next call and execute an actual recursive call with the call-state for that node pair. After all recursive calls finished, we remove the element from the path. It is possible that during a recursive call on a lower level work packages have been generated for the remaining node pairs. In this case, the path will be empty when the recursive call returns. We check for this, and then return directly instead of continuing with the remaining node pairs.

For our recursive calls, we also update a copy of the graph, the blocked node pairs and the subgraph counters that are used for calculating lower bounds and the branching strategy. Additionally to this copy that represents the state at the *bottom* of our path, we also maintain a copy that corresponds to the state at the *top* of the path that is used for generating work packages.

To generate work packages, we first advance the *top* state to the next node pair that has not been used for a recursive call. For all remaining node pairs of the top element of the path we generate a separate work package using the top state and the call-state that is stored in the path's element. Then, we remove the top of the path. This continues with the new top of the path, until either the recursion path is empty or a sufficient number of work packages ($2x$ number of threads) are in the queue.

Note that we generate work packages before creating the element in the path that corresponds to the current recursive call. Hence, the generating thread still has work left, even if the recursion path becomes empty. This is to avoid that a thread immediately needs to get another work package after putting work into the global queue.

Part IV

Local Communities

11 Local Community Detection Based on Small Cliques

This chapter is based on joint work with Eike Röhrs and Dorothea Wagner [HRW17]. Compared to the publication, this chapter has been slightly adapted to reference parts of this thesis instead of repeating introductions. Further, experiments have been repeated due to several bugs in the original implementation. While the new results support our conclusions, there are slight differences that are discussed in this chapter.

The idea of local community detection is to detect a community around a seed node without considering the whole graph. If only communities of few nodes are required, this allows these community detection algorithms to scale to arbitrarily large graphs while being potentially much faster. Further, many overlapping community detection algorithms use local community detection algorithms as basic building block. We consider the problem of finding a community locally around a seed node both in unweighted and weighted networks. We provide a broad comparison of different existing strategies of expanding a seed node greedily into a community. For this, we conduct an extensive experimental evaluation both on synthetic benchmark graphs and real world networks. We show that results both on synthetic and real-world networks can be significantly improved by starting from the largest clique in the neighborhood of the seed node. Further, our experiments indicate that algorithms using scores based on triangles outperform other algorithms in most cases. We provide theoretical descriptions as well as open source implementations of all algorithms used.

11.1 Introduction

In this chapter, we consider local community detection where a single community around a given seed node or a set of seed nodes shall be detected. Local methods can be applied when the user is, e.g., just interested in identifying a group of people around a given person in a social network or the functional subset a molecule belongs to within a biochemical network, and thus a clustering of the whole network is not needed. The advantage of local methods is that their performance usually just depends on the size of the detected community and not on the size of the whole network, which might be very large or even unavailable. An example for such a network is the network that is defined by the pages of the web and the links between them. Accessing this network by starting at a single node is easy and computationally feasible. Having access to the whole network requires huge computational resources and is basically impossible, so local community detection algorithms provide a viable alternative here. Various local methods have been proposed that use the structure of the network to detect communities locally [SMM14; Sch07]. The study in [SMM14] however shows that local algorithms still perform significantly worse than global, disjoint community detection algorithms on synthetic benchmark graphs. Expanding single nodes (LFM [LFK09], OSLOM [Lan+11]), single edges (MOSES [MH10]) or maximal cliques (GCE [Lee+10]) into communities is nevertheless a successful scheme for overlapping community detection.

In this chapter, we compare different measures and algorithms for expanding communities by greedily adding (and possibly removing) single nodes. Greedy expansion is both a common and usually also scalable approach for finding communities. In our extensive experimental evaluation we show that in many cases the seed node is not embedded in the detected community. The reason for this is that many algorithms fail to correctly select the first nodes to add. To avoid this problem, we examine starting the expansion of the community from a maximum clique that contains the seed node as proposed by [Fan+14]. Further, we introduce Triangle Based Community Expansion as an alternative strategy for greedy community expansion. It exploits the fact that edges inside communities are usually embedded in triangles [Rad+04] and uses an edge score based on triangles for deciding which node to try next.

We compare the performance of the algorithms on synthetic benchmark graphs generated using the LFR benchmark [LF09b; LF09a] with unweighted and weighted graphs with disjoint ground truth communities as well as unweighted graphs with overlapping ground truth communities. Further we also perform experiments on real-world social networks. We implemented all examined algorithms in the network analysis toolkit NetworKit [SSM16] and make them publicly available [1], we aim to make them part of the official NetworKit code base. We show that starting from a clique dramatically improves the accuracy of the detected communities both on synthetic and on real-world networks. In the experimental evaluation, we show that Triangle Based Community Expansion has a solid performance and even without starting from a clique it still often finds the correct community. The computationally more expensive LTE algorithm [Hua+11] whose scoring is based on triangles is among the best-performing algorithms on most tested graphs and has the best performance on the tested real-world social networks. This indicates that exploiting local structures like triangles is important for finding communities without global knowledge and that exploiting such structures might lead to the development of better overlapping clustering algorithms. To the best of our knowledge, we are the first to conduct such a systematic and broad study that shows that starting with cliques does not only improve the detected community, but also makes sure that the chosen seed node is actually embedded in the detected community, an aspect frequently ignored in existing evaluations.

In this chapter, we treat all graphs as weighted graphs. We use the notation as introduced in Section 1.2. In the next section, we provide a brief overview of the related work. In Sections 11.2 and 11.3 we present in more detail the algorithms we compare. Thereafter, we show our experimental results. Lastly, we conclude with a summary and outlook in Section 11.5.

[1] `https://github.com/kit-algo/LCD-cliques-networkit`

11.1.1 Related Work

Numerous local community detection algorithms have been proposed [SMM14; Cla05; CZG09; Bag08; FZB14]. Many of these algorithms can be grouped as greedy community expansion [SMM14], which also provides the basis of other (more complex) algorithms. It can be described best as greedily adding the node to the community that currently has the maximum value of some function that assigns a value to each node. This greedy community expansion usually stops once the value of every node that could still be added is negative. Often measures based on the ratio of internal edges to edges that are connected to the rest of the graph are used. BME1 [NTV12] additionally uses the distance to the seed node to assign less importance to edges from nodes further away from the seed node. Other algorithms like iGMAC [Ma+14] also consider a wider neighborhood of the nodes to be added and are thus computationally more expensive.

However, there are also algorithms that work with a different strategy, which is not based on greedy community expansion. One of these is PageRank-Nibble, which works by locally approximating PageRank-vectors [ACL06]. It first computes a ranking of the nodes and only then finds a community based on this ranking. Flow-Pro [PPF15] starts a network flow from the seed node and adds nodes to the communitiy when a sufficient part of the flow arrives at them. LEMON is an approach based on local spectral optimization [Li+15]. From these algorithms, we only include PageRank-Nibble in our evaluation as for the other algorithms no efficient implementation is available.

Many of the above-mentioned algorithms can also start from a set of seed nodes instead of a single seed node. This can be used to identify a community more precisely as discussed in [DGG13]. Starting from a maximum clique as proposed in [Fan+14] is a similar idea that helps identifying the community more precisely.

Triangles are not only an important part of our Triangle Based Community Expansion algorithm and the LTE algorithm [Hua+11], but are also used in other community detection algorithms [FZB14; Jia+14; Rad+04]. Triangles have also already been popularized with respect to social networks by the social sciences with the term of triadic closure and are the basis of the clustering coefficient [WS98; EK10].

11.2 Density-Based Local Community Detection Algorithms

In this section, we introduce the existing community detection algorithms based on the density of internal and/or external edges of the community. We will compare these algorithms later in the experimental evaluation. For some of them we describe how we adapted them for weighted graphs and for most of them we describe how their running time can be optimized, an aspect often not included in the original publications. If the algorithms have any parameters, we also mention the values we choose for the experiments. We use the same values throughout all experiments as tweaking the parameters for each graph (or set of graphs) requires knowledge about the expected communities which is not always available in practice. For this chapter, we only selected algorithms that are local in the sense that they use only information about nodes located around the seed node, and, except PageRank-Nibble, do not assume knowledge about the whole graph. In particular, this also means that these algorithms do not know the number of nodes or edges and their running time does not depend on them.

11.2.1 GCE M/L

One of the simplest schemes of local community detection is expanding a community by adding in each step the node of the shell that leads to the highest quality improvement of the community. In each step, the next node is selected by iterating over all nodes in the shell of the community. We examine the two quality measures proposed for this algorithm by [SMM14]: The M measure [LWP08] and the L measure [CZG09].

The M measure is defined as the number of internal edges of the community divided by the cut of the community. This can be trivially generalized to weighted graphs by taking the sum of the edge weights instead of the number of edges:

$$M(C) := \frac{\mathrm{vol}_w(C) - \mathrm{cut}_w(C)}{2 \cdot \mathrm{cut}_w(C)}$$

Large values of the M measure indicate internally dense and well-separated communities. It was shown [SMM14], that maximizing the M measure is actually equivalent to minimizing the conductance of the community as long as the volume of the community is smaller than the volume of its complement. The *conductance* of a community C is defined as:

$$\textsc{conductance}(C) = \frac{\mathrm{cut}_w(C)}{\min(\mathrm{vol}_w(C), \mathrm{vol}_w(V \setminus C))}. \tag{11.1}$$

The L measure is defined using two terms, an internal and an external part. The internal part L_{in} is defined as the internal edge weights divided by the size of the community:

$$L_{in} := \frac{\mathrm{vol}_w(C) - \mathrm{cut}_w(C)}{|C|}$$

The external part L_{ex} is defined as the cut of the community divided by the size of the boundary:

$$L_{ex} := \frac{\mathrm{cut}_w(C)}{|B(C)|}$$

The L measure is then the internal divided by the external L measure, i.e., $L := L_{in}/L_{ex}$.

Note that if a community has no internal nodes, i.e., all nodes are adjacent to at least a node outside the community, this is actually equivalent to the M measure as then $|C| = |B(C)|$. Therefore, large values of the L measure again indicate internally dense and externally well-separated communities.

Implementation Details

To expand the community, we need to determine which node of the shell will lead to the maximum improvement of the quality measure. In order to allow to compute this in constant time per node in the shell, we store for every node in the shell the sum of the edge weights towards the community and the sum of the remaining edge weights. This allows to determine the best candidate for the M measure in a single scan over the shell. Whenever we add a node to the community, we iterate over its neighbors and update their values. When a node is not yet in the shell, we also need to determine its initial external degree, which is its degree minus the weight of the edge through which we discovered the node. For unweighted graphs our graph data structure already stores the degree so this is possible in constant time, for weighted graphs we need to iterate over the neighbors to determine the weighted degree. Therefore, for unweighted graphs we get a total running time of $\mathcal{O}(\mathrm{vol}(C) + |C| \cdot |S|)$, where C is the final community and S is the largest shell encountered during the execution of the algorithm. For weighted graphs, we get an additional running time of $\mathcal{O}(\mathrm{vol}(S(C)))$ if the weighted degrees are not pre-computed.

For the L measure we additionally need to be able to tell how much the boundary will change when we add a node u of the shell to the community C. This depends on two factors: (a) if u has any external neighbors, the boundary size is increased by 1, (b) if u has any neighbors in C whose only external neighbor is u, then these nodes will no longer be part of the boundary. The first factor can be easily determined by looking at the external degree of u. For the second factor, we maintain two counters: For every node u in the boundary we store how many external neighbors $\mathrm{ex}(u)$ it has. For every node v in the shell we store the counter $\mathrm{exin}(v)$ that denotes how many neighbors of v in C have only v as external neighbor, i.e., the number of nodes $x \in N(v) \cap C$ with $\mathrm{ex}(x) = 1$. The counter exin allows us to determine the change in the boundary size in constant time. We update these counter values as follows: Whenever we add a node u to the community, we decrement the number of external neighbors $\mathrm{ex}(v)$ of every neighbor v of u by 1. When the number of external neighbors $\mathrm{ex}(v)$ for a node v in the boundary drops to 1, we need to notify this external neighbor $x \in N(u) \setminus C$ as their counter $\mathrm{exin}(v)$ needs to be increased. To determine x, we iterate over all neighbors of v. We then increment $\mathrm{exin}(v)$ by 1. Note that the number of external neighbors $\mathrm{ex}(v)$ only decreases as more nodes are added to the community. Therefore, this will happen only once for every node in the boundary. The asymptotic running time is thus the same as for optimizing the M measure.

11.2.2 Two-Phase L

The originally proposed algorithm for L-based community detection [CZG09] is slightly more complicated than GCE L. It considers three cases for the updated measures L'_{in} and L'_{ex} that may lead to a higher value L':

1. $L'_{in} > L_{in}$ and $L'_{ex} < L_{ex}$

2. $L'_{in} < L_{in}$ and $L'_{ex} < L_{ex}$
3. $L'_{in} > L_{in}$ and $L'_{ex} > L_{ex}$

In a first discovery phase, they iteratively add the node that leads to the maximum L', but only if it is in case 1 or 3, otherwise they do not consider that node again. In a second examination phase, they re-consider every node and keep it only when it is in the first case. It may happen that the original seed node is removed from the community in the second phase. Then they say that there is no local community for that seed node.

Implementation Details

For the first discovery phase, we can use the same data structures as described for GCE L. In the second examination phase, we can simply iterate over the neighborhood of every node again to determine the change in L'_{in} and L'_{ex} its removal would cause. This means in total we get the same running time as GCE L, though the running time depends on the community size after the first phase, not on the final size.

11.2.3 LFM

The LFM algorithm [LFK09] maximizes the fitness function

$$f(C) := \frac{\operatorname{vol}_w(C) - \operatorname{cut}_w(C)}{\operatorname{vol}_w(C)^\alpha}$$

where α is a real-valued, positive parameter. In each iteration, it adds the node that leads to the highest improvement of $f(C)$ to the community C and then removes every node whose removal increases $f(C)$. The parameter α can be used to influence the size of the detected communities. Large values of α correspond to small communities while small values correspond to larger communities. The authors report [LFK09] that values below 0.5 usually lead to just one community while values larger than 2 led to the smallest communities. Further, they recommend $\alpha = 1$ as a natural choice which is also the parameter we use in our experiments.

Their proposed LFM algorithm is actually an algorithm for detecting overlapping communities that cover the whole graph by iteratively expanding a random seed node that has not yet been assigned to a community into a new community. We only consider the basic building block of local expansion here.

Implementation Details For LFM, we use the same data structure as GCE M for adding nodes to the community. In order to allow removals by simply scanning over the nodes of the community once, we also store internal and external (weighted) degrees of all nodes in the community. The running time is not so easy to state in terms of the community size as nodes may be added and removed again. Each addition of a node u needs time $\mathcal{O}(\deg(u) + |C| + |S(C)|)$ as we scan first the shell to find u and then the community to find a possible candidate for removal. Each removal of a node u needs time

$\mathcal{O}(\deg(u) + |C|)$ as we need to scan C again to find other nodes that possibly need to be removed. Further, every time we add a node to the shell we might also need to calculate its total weighted degree unless it has been pre-computed.

11.2.4 PageRank-Nibble

PageRank-Nibble [ACL06] is a two-step algorithm: first, it approximates personalized PageRank vectors around the seed node and sorts all nodes with a positive score according to that score in decreasing order. Then it considers all communities that are a prefix of this sorted list and returns the prefix with minimum conductance as community. A larger study of using different measures with the same ordering criteria shows that conductance is actually a good choice [YL15]. The approximation of the personalized PageRank vectors needs two parameters that determine the accuracy of the approximation (ϵ) and the loop probability of a random walk (α). We use the values $\epsilon = 0.0001$ and $\alpha = 0.1$, these are the values that performed best in a parameter study in [SMM14]. The running time of the algorithm is in $\mathcal{O}((\epsilon \cdot \alpha)^{-1})$, i.e., for given parameter values, it is constant and therefore the maximum detected community size is actually also constant. Finding the prefix with the best conductance is possible in time linear in the volume of the found nodes if the total volume of the graph is known, otherwise this needs to be computed to be able to optimize the true value of conductance and the algorithm is not actually a local community detection algorithm.

11.3 Local Community Detection Algorithms Based on Small Cliques

When the GCE or LFM algorithm starts, the community consists only of a single node. Regardless which neighbor is added first, the internal density of the resulting community of two nodes is the same. Therefore, regardless of the exactly used measure, the neighbor of the lowest degree will be preferred. If the two nodes do not have a common neighbor, again the neighbor of the lowest degree is chosen. This means that at the beginning of the expansion phase, nodes of low degree are preferred regardless if they actually belong to the community of the seed node. Therefore, it is possible that the community grows into a community where the seed node is not strongly embedded in. Experiments (which are reported at the end of Section 11.4.3) show that this is actually a problem.

A possibility to avoid such problems is to consider not only the density of the community but also its internal structure and the structure of its neighborhood. Not all edges are equally embedded in their neighborhood, some of them have more common neighbors than others. The common neighbors of the two nodes of an edge thereby form a triangle, i.e., a clique of size three. In Chapter 7, we already explored the use of scores based on triangles to filter edges that are not contained in communities and showed that this can amplify the community structure. Here, the same idea is used to identify edges that are likely inside a community. The local tightness expansion algorithm (LTE) [Hua+11] uses an edge similarity score based on triangles both for the decision which node to add next and for the quality of the community. LocalT [FZB14] uses a quality measure for the community expansion process that is directly based on the number of triangles in and outside of the community.

A different approach is to start directly with the maximum clique in the subgraph induced by the neighbors as proposed in [Fan+14]. This also avoids the first error-prone steps. Our own approach, Triangle Based Community Expansion (TCE) is similar to LTE, we also use an edge score based on triangles. However, for the quality function of the community we use conductance. In the following, we present all four approaches.

11.3.1 LTE

The local tightness expansion algorithm (LTE) [Hua+11] defines its similarity score s of two connected nodes u, v as follows:

$$s(u,v) := \frac{2 \cdot w(u,v) + \sum_{x \in N(u) \cap N(v)} w(u,x) \cdot w(v,x)}{\sqrt{(1 + \sum_{x \in N(u)} w^2(u,x)) \cdot (1 + \sum_{x \in N(v)} w^2(v,x))}}$$

They then define the *tightness* $T(C)$ of a community as

$$T(C) := \frac{S_{in}^C}{S_{in}^C + S_{out}^C},$$

where S_{in}^C is two times the sum of all internal similarities, i.e., using s as weight function, $S_{in}^C = \mathrm{vol}_s(C) - \mathrm{cut}_s(C)$ and S_{out}^C is the sum of similarities between adjacent nodes inside and outside the community, i.e., $S_{out}^C = \mathrm{cut}_s(C)$.

LTE always tries to add the node with "the largest similarity with nodes in C" [Hua+11]. We interpret this as the sum of the similarities of the node and all adjacent nodes in C. This set of candidates can be efficiently maintained in a priority queue. A node is only added to the community if its tunable tightness gain is positive. This tunable tightness gain of a node a is defined as

$$\tau_C^\alpha(a) := \frac{S_{out}^C}{S_{in}^C} - \frac{\alpha S_{out}^a - S_{in}^a}{2 S_{in}^a},$$

where S_{in}^a is the sum of the similarities of a and all adjacent nodes in C and S_{out}^a is the sum of the similarities of a and all adjacent nodes outside of C. When $\tau_C^\alpha(a)$ is not greater than 0, a is removed from the set of candidates. If a is added to the community, the set of candidates is updated with the neighbors of a, therefore, a node may be re-added to the queue later if a neighbor of it is added to the community.

Implementation Details The original paper [Hua+11] does not further address the calculation of the edge similarity scores but simply assumes they are available in constant time. We do not assume this but instead use an efficient triangle listing algorithm to calculate them. To allow fast triangle listing, we build and maintain a local graph data structure. For processing the whole queue, we need to know the external similarity for every node in the shell and therefore we need to be able to list all triangles a node in the shell is part of. To allow this, our local graph H is the subgraph induced by the community C, its shell $S(C)$ and the shell of the shell $S(S(C))$. Using a hash table, we map global node ids to local, small integer node ids.

Note that when a new node u is added to the community, only scores of neighbors of u are affected. Therefore, we need to either add u's neighbors that are not in C to S or update their scores. The former means that the score of node v only consists of $s(u, v)/\deg_w(v)$. The latter implies that we need an $\textsc{increaseKey}$ operation with the updated score: $\textsc{score}_{\text{new}}(v) = \textsc{score}_{\text{old}}(v) + s(u, v)/\deg_w(v)$. The total running time of all priority queue operations are therefore $\mathcal{O}(\text{vol}(C)\log(n_H))$ where n_H denotes the number of nodes in H. Using the h-graph data structure presented in [LSS12], the triangle listing is possible in total time for all nodes $\mathcal{O}(m_H \cdot \alpha(H))$, where m_H is the number of edges in H and $\alpha(H)$ is the arboricity of H. As we do not need all operations, we only maintain the list of higher-degree neighbors of every node and therefore only need a simplified version of the h-graph data structure. Note however, that, in order to add a node to H, we need to check for all of its incident edges if the endpoint is in H. Therefore, in our situation we get an additional expected running time of $\mathcal{O}(\text{vol}(C \cup S(C) \cup S(S(C))))$, where vol denotes the volume in the whole graph. Whenever we add a node u to the community C, we list all triangles u is part of and update the scores of its neighbors. We then also add all its neighbors to the shell and for each of them we list all their triangles to compute the initial external score values.

For storing the shell, we use a 4-ary (max-)heap that supports increasing keys. This allows us to update scores whenever we add a node to the community C. In total, we need $\mathcal{O}(\text{vol}(C))$ priority queue operations yielding total costs of $\mathcal{O}(\text{vol}(C)\log(\text{vol}(C)))$.

The total running time is composed of building the local graph H, computing the scores by listing triangles and updating the priority queue, which is in $\mathcal{O}(\text{vol}(C \cup S(C) \cup (S(S(C)))) + m_H \cdot \alpha(H) + \text{vol}(C)\log(\text{vol}(C)))$. Studies on real world networks [LSS12] show that the arboricity is small in practice, which means that the running time is close to being linear in the volume of the nodes of H in the original graph.

If the whole input graph shall be covered by communities, the scores can be precomputed. The running time is then reduced to the required time for the priority queue operations.

11.3.2 Local T

The local T algorithm [FZB14] optimizes their T-measure which is based on internal and external triangles of the community. They define T_{in} as the number of triangles where all three nodes are in C. The number of external triangles T_{ex} is then the number of triangles with exactly one node in C. Note that triangles with exactly two nodes in C are neither internal or external according to this definition. The T metric is then defined as follows:

$$T := T_{in} \cdot \begin{cases} T_{in} - T_{ex} & \text{if } T_{in} \geq T_{ex} \\ 0 & \text{otherwise} \end{cases}$$

Their approach is also iterative, in each iteration the node that maximizes the T-measure is added to the community. If there are ties, the node with the lowest T_{ex} score is preferred. Additionally, they detect outliers, i.e., nodes only weakly connected to a community, and hubs. We do not evaluate these steps here as identifying outliers or hubs is not the goal of this work.

Implementation Details Similar to the data structure for LTE, we also maintain a local graph for Local T. The main algorithmic difference is that no priority queue is used but instead for every node to be added to the community, the whole shell is scanned. Apart from that the algorithm is very similar, we list triangles and update scores at exactly the same places. Therefore, the total running time is $\mathcal{O}(\mathrm{vol}(C \cup S(C) \cup (S(S(C)))) + m_H \cdot \alpha(H) + |C| \cdot |S|)$, where $|S|$ is the maximum size of the shell which is in $\mathcal{O}(\mathrm{vol}(C))$.

Note that for LocalT each triangle needs to be considered in the context of the community and therefore no scores can be pre-computed to make the algorithm faster.

11.3.3 Clique Based Community Expansion

The basic idea of Clique Based Community Expansion is to start with the maximum clique in the subgraph induced by the neighbors of the seed node. For weighted graphs, we use the clique that has the maximum sum of all edge weights inside it and towards s. This is an extension that can be applied to all algorithms we consider. For algorithms based on simple greedy addition of nodes, we simply first add all nodes of the clique, for PageRank-Nibble the PageRank approximation simply starts with the initial weight equally distributed to the nodes in the clique.

For overlapping community detection, this was already considered by the Greedy Clique Expansion algorithm [Lee+10]. It first lists all maximal cliques in the graph and then expands a subset of them into communities. For local community detection, this was proposed by [Fan+14]. While they state that this can be combined with any greedy expansion scheme, they only evaluate it in combination with GCE M.

Implementation Details Our Clique Based Community Expansion implementation first creates the subgraph induced by the neighbors $N(s)$ of the seed node s. Then we detect cliques using the simple ELS-bare algorithm of Eppstein et al. [ELS13], which runs in time $\mathcal{O}(d^2 n 3^{(d/3)})$ on a graph with n nodes and degeneracy d. While they also present a faster algorithm, this algorithm still achieves good running times in practice while being much simpler to implement [ELS13]. In our usage of the algorithm $n = \deg(s)$, since we only look in the neighborhood of a seed node s. For unweighted graphs we simply select the largest clique, for weighted graphs we select the clique of maximum edge weight. For the weighted case, we simply iterate over all neighbors of the clique nodes and sum up the weights of intra-clique edges. As a graph with degeneracy d has at most $d(n - \frac{d+1}{2})$

edges and the number of maximal cliques is at most $(n-d)3^{(d/3)}$ [ELS10], this step needs at most $\mathcal{O}(d \cdot n^2 \cdot 3^{(d/3)})$ asymptotically. Therefore, the total running time of the initialization algorithm for unweighted graphs is the same as the running time for finding the cliques on the graph induced by the neighbors of the seed node, while for weighted graphs the clique selection step dominates the asymptotic worst case running time.

11.3.4 Triangle Based Community Expansion

Our Triangle Based Community Expansion (TCE) algorithm maintains the shell of the community in a priority queue. The priority queue is sorted by a node score that determines how well a node is connected to the community, taking the neighborhood of the node and its neighbors into account. Our algorithm works iteratively by extracting a node from the queue and, if it improves the quality of the community, adding it to the community. If it is not added, we proceed with the node that has the next highest score and continue until no node from the current shell meets the condition.

Our node score uses an edge score based on triangles. We chose triangles as they were shown to be a good indicator of community structure and are successfully used for community detection [YL15; Jia+14; FZB14]. Specifically, we found the edge score described by Radicchi et al. [Rad+04] to perform quite well for unweighted graphs. The score is given by the following equation:

$$\frac{|N(u) \cap N(v)| + 1}{\min(\deg(u) - 1, \deg(v) - 1)} \tag{11.2}$$

We adapt this score to weighted graphs by replacing the +1-term by the weight of the edge and the size of the intersection by the minimum weight of the two involved edges. Further, we removed the −1 in the denominator as it is otherwise not clear what the score should be when one of the nodes has only one neighbor. Therefore, our edge score is given by the following equation:

$$\omega(u,v) = \frac{w(u,v) + \sum_{x \in N(u) \cap N(v)} \min(w(u,x), w(v,x))}{\min(\deg_w(u), \deg_w(v))}. \tag{11.3}$$

It can be understood as the ratio between the number/weight of actual triangles and the maximum possible number/weight of triangles. Note that for strictly positive edge weights, this edge score is also always strictly positive and at most 1.

The node score is computed for a node u as:

$$\textsc{score}(u) = \frac{1}{\deg(u)} \sum_{v \in N(u) \cap C} \omega(u,v), \tag{11.4}$$

which is the sum of all edge scores from edges connecting the node u to the community C, divided by its unweighted degree. We normalize the score by the node's unweighted degree, so that nodes with high degree do not have high scores, simply by virtue of having many edges connecting it with the community, regardless of edges, that connect it to other parts of the graph. Due to this normalization, node scores are also at most 1.

```
Input: Graph $G = (V, E)$, edge weights $w(\cdot, \cdot)$, seed node $s$.
Output: A community $C$ of the seed node $s$.
1 $C \leftarrow \{s\}$
2 $S \leftarrow N(s)$
3 while $S \neq \emptyset$ do
4     $u_{\max} \leftarrow \arg\max_{u \in S} \textsc{score}(u; G, C, w)$
5     $S \leftarrow S \setminus \{u_{\max}\}$
6     if $\textsc{conductance}(C \cup \{u_{\max}\}) < \textsc{conductance}(C)$ then
7         $C \leftarrow C \cup \{u_{\max}\}$
8         $S \leftarrow S \cup (N(u_{\max}) \setminus C)$
```

Algorithm 11.1: Triangle Based Community Expansion (TCE) detects a community around a given node. Uses a node scoring function based on triangles to add nodes to the community.

By always removing the node with the maximum score from the shell, we consider adding every node in the shell to C at some time. However, using the node scores we prioritize some nodes over others, which leads to community growth in the directions of nodes with higher node scores.

We add a node to the community, when adding it to the community improves the conductance of the community. We choose conductance as it intuitively captures the definition of a community and was shown to perform better than other scores in comparative study [YL15]. To use the original conductance score as shown in Equation (11.1), we need to know $\text{vol}(V \setminus C)$ in order to properly compute the conductance of a community C, which requires knowledge of the entire graph's volume. As we do not want to assume this, we always use $\text{vol}(C)$ in the denominator of conductance in Equation (11.1). This approach mostly yields the same values as the original equation, since the community is usually small in relation to the whole graph.

We show the pseudo-code of our algorithm TCE in Algorithm 11.1. It takes a graph G with edge weights w and a seed node s. For simplicity, we assume access to the whole graph here, however the algorithm does not actually access the whole graph but only a local neighborhood of the nodes added to the community. It begins by initializing the community C with s and the shell S as $N(s)$, the neighbors of s (lines 1,2). Then the loop is executed, taking the node $u_{\max} \in S$ with the highest node score (line 4), which is given by Equation (11.4). This node is then removed from S (line 5) and we check if it would improve the conductance of C. If it improves the conductance, we add the node to C and update S by inserting all neighbors of $u_{\max}$ that are not in C (lines 6–8).

Implementation Details As for LTE and Local T, we build and maintain a local graph data structure for counting triangles. For TCE, we only need the subgraph H induced by the community C and its shell $S(C)$, as we are only interested in triangles where at least one node is in C. Like for LTE, we use a 4-ary (max-)heap for storing S that supports increasing keys.

When a new node u is added to the community, only scores of neighbors of u are affected. Therefore, when a node u is added to the community, we only need to either add u's neighbors that are not in C to S or update their scores (Algorithm 11.1 (line 8)). The former means that the score of node v only consists of $\omega(u,v)/\deg_w(v)$. The latter implies that we need an INCREASEKEY operation with the updated score: $\textsc{score}_{\text{new}}(v) = \textsc{score}_{\text{old}}(v) + \omega(u,v)/\deg_w(v)$. The total running time of all priority queue operations are therefore $\mathcal{O}(\text{vol}(C)\log(\text{vol}(C))$.

The total running time is composed of building the local graph H, computing the scores by listing triangles and updating the priority queue. In total, this is $\mathcal{O}(\text{vol}(C \cup S(C)) + m_H \cdot \alpha(H) + \text{vol}(C)\log(\text{vol}(C)))$. Note that compared to LTE, the subgraph H is smaller as it does not contain the neighbors of the shell.

11.4 Experiments

We compare the previously presented algorithms including the initialization using a clique on weighted and unweighted graphs. Firstly, we present our experimental results on synthetic graphs from the LFR benchmark [LF09b; LF09a]. Secondly, we evaluate our algorithms on 100 Facebook friendship networks [TMP12]. We start by describing our experimental setup and the used scoring and then continue with the results on the synthetic and the real-world networks. At the end of this section, we also evaluate the running time of the different algorithms.

11.4.1 Experimental Setup

For all of these local community detection algorithms, we provide novel or improved implementations in C++ as part of the open source network analysis tool suite NetworKit [SSM16]. The exact implementations used and the scripts used to generate graphs, evaluate the detected communities and generate the plots are available online [2]. We aim to contribute the implementations to NetworKit.

We also compare against the global community detection algorithm Infomap [RAB09] (see Section 1.3.4 for an introduction) which has been shown to perform excellently on the synthetic benchmark graphs we use [LF09b]. This allows us to see how a state-of-the-art global community detection algorithm performs in comparison to the local community detection algorithm. For Infomap we use the implementation provided by the authors to optimize a two-level partition, i.e., a disjoint clustering of the whole graph. To get the community of a seed node, we simply use the cluster detected by Infomap that contains the seed node.

All results are obtained on a computer with an Intel® Core™ i7 2600K Processor, run at 3.40 GHz with 4 cores, activated hyper-threading, and 32 GB RAM.

We generate 20 random realizations for each synthetic graph and select a set of 20 random seed nodes for each of them. In our experiments on real world networks we use

[2] `https://github.com/kit-algo/LCD-cliques-experiments`

100 randomly chosen nodes per graph as seeds, as there we have only one realization per graph.

11.4.2 Scoring

We compare the detected communities to the ground truth communities generated in the synthetic benchmark graphs and the communities induced by the attribute “dormitory” in the Facebook 100 dataset. For simplicity, we will call them also “ground truth” communities in the following. For the comparison, we match each detected community with a ground truth community. We use two variants of this: one where we compare the detected community to the best-matching ground truth community that contains the seed node and one where we choose the best-matching ground truth community irrespective of the seed. As score, we use the standard F_1 score as introduced in Section 2.3.3. The F_1 score is between 0 and 1, where 1 is the best score, it indicates a perfect match of ground truth and detected community.

The F_1-score and the similar Jaccard-Index are often-used measures in the area of community detection and graph clustering [YL14; SMM14; Ma+14; CZG09]. Both work well when comparing a single community that an algorithm finds to one or several ground truth communities by taking into account nodes correctly and incorrectly identified. Many other comparison measure typically used for comparing detected communities to ground truth communities like NMI assume a whole clustering consisting of many communities covering the whole graph. They are thus not suitable for our evaluation where we explicitly want to compare individual detected communities to (possibly only some) ground truth communities.

In order to give an idea how good a given F_1 score is, we implemented a simple baseline algorithm. The simplest idea would be to return a random set of nodes. However, any connected subgraph containing the seed node could easily outperform this. To get a more realistic, yet basically random community we instead perform a breadth-first search starting at the seed node. As input, it takes not only the seed node but also the size k of a random ground truth community the seed node is part of. It returns the first k nodes it finds during the breadth-first search. Therefore, the size of the detected community always perfectly matches the size of one of the grount truth communities the seed node is part of. In the experiments, we denote this algorithm by “BFS”.

11.4.3 Synthetic Graphs

We use the LFR synthetic benchmark graphs as described in the introduction in Section 2.1.1. We use the variants with unweighted graphs with disjoint cluster as well as overlapping clusters and weighted graphs with non-overlapping clusters. We summarize the parameters and our chosen values in Table 11.1.

For unweighted, disjoint communities we use the implementation of the LFR benchmark in NetworKit [SSM16]. For the weighted and overlapping variants we use the implementations provided by the authors of the LFR benchmark.

Table 11.1: Parameters of the LFR benchmark that we use in our experiments.

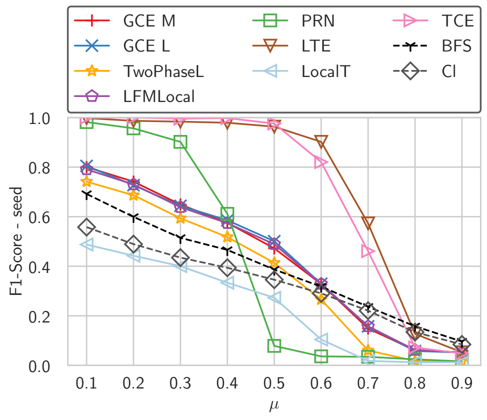
Name	**Description**	**Unweighted**	**Overlapping**	**Weighted**
n	number of nodes	5000	2000	5000
k	average degree	20	39.5, 61.5, 78.1, 91.8, 103.5	20
$k_{\max}$	maximum degree	50	120	50
τ_1	degree exponent	-2	-2	-2
$C_{\min}$	minimum community size	10, 20	60	20
$C_{\max}$	maximum community size	50, 100	120	100
τ_2	community size exponent	-1	-2	-1
μ_t	topological mixing	0.1, ..., 0.9	0.2	0.3, 0.5, 0.8
β	weight exponent			-1.5
μ_w	weight mixing			0.1, ..., 0.9
O_m	communities per node	1	1, ..., 5	1

Unweighted Graphs

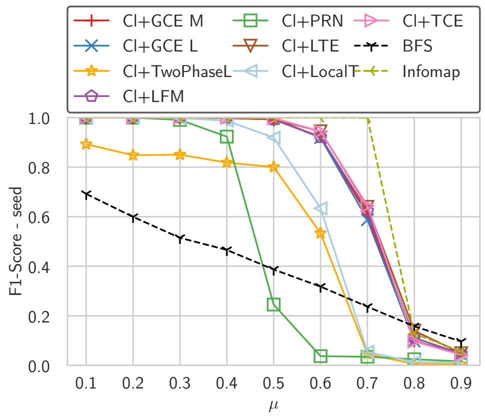
For the first experiment, we run the algorithms on the parameter set Unweighted in Table 11.1, which are the large (5000 node) variants used by Lancichinetti et al. [LF09b]. This parameter set consists of two different community size ranges [10, 50] and [20, 100]. Therefore, we generate two different sets of graphs for which we vary μ_t, the topological mixing parameter. We generate 20 instances of each graph and let the algorithms run from 20 randomly picked seed nodes per graph.

Figure 11.1 shows the average F_1-scores both for the variants starting from a single seed node (left column) and the variants starting from the maximum clique (right column). The "Cl" algorithm in the left column is just the maximum clique of the seed node, i.e., the starting point of the algorithms in the right column.

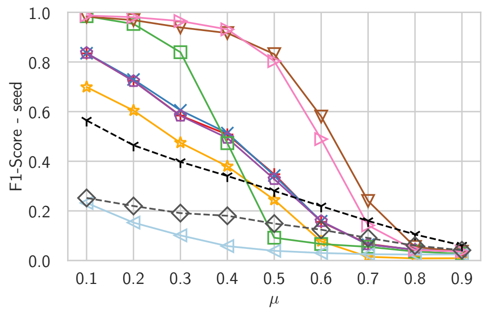
The performance of the plain Local T algorithm is well below the performance of the baseline BFS. Even the simple clique algorithm outperforms it in terms of F_1 score. With clique initialization it performs better but is still one of the lowest-scoring algorithms. Note that due to a bug in the handling of the initialization with multiple seed nodes, LocalT performed worse in our original evaluation [HRW17]. The density-based simple greedy expansion algorithms GCE M, GCE L and LFM all perform very similar, which is not unexpected as they are based on very similar quality functions. With a clique as initialization their performance is significantly improved, in particular on the smaller communities they return almost perfect communities even past $\mu = 0.5$. The original L-based community detection algorithm, Two-Phase L, performs worse than these simple expansion algorithms. One explanation for this behavior is that if the seed node is no longer part of the detected community, the algorithm simply returns an empty community, giving a score of 0. Only the triangle-based algorithms LTE and TCE perform well when initialized with a single seed node. They also profit from the initialization with a clique,



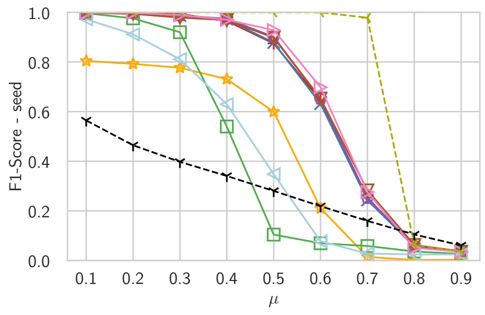
(a) $n = 5\,000$, small



(b) $n = 5\,000$, small



(c) $n = 5\,000$, big



(d) $n = 5\,000$, big

Figure 11.1: Avg. F_1-scores on the LFR benchmark with the parameter set Unweighted, which we specify in Table 11.1. The left column shows results when starting with a single seed node, the right column shows results for starting with the maximum clique as well as Infomap for comparison.

though. In particular on the smaller communities their performance is not far from the Infomap algorithm, which is able to perfectly recover the community structure till $\mu = 0.7$. The PageRank-Nibble algorithm (PRN) performs only well on the graphs with low values of μ, for higher values its performance degrades very quickly.

To explore the performance of the different algorithms in more detail and get some explanations for their behavior, we show more detailed results in Figure 11.2. In Figure 11.2a,b, we see the conductance of the detected communities. Due to the definition of conductance and the ground truth communities in LFR, μ corresponds exactly to the conductance of the ground truth communities. We can see here that the triangle-based algorithms which almost perfectly recover the community structure (TCE and LTE) also yield good, though not perfect conductance values. For the methods based on density and simple greedy expansion, GCE M, GCE L, LFMLocal, we can see that the coductance of the detected communities is improved by starting from a clique.

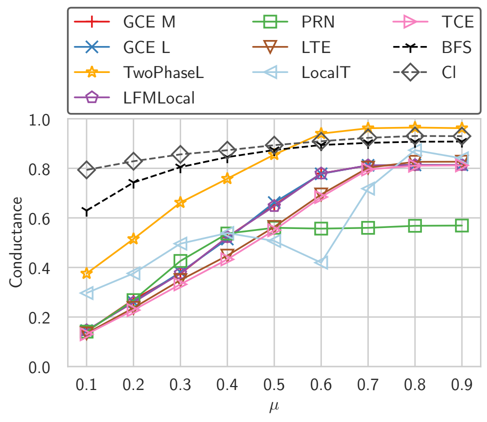
PageRank-Nibble shows an interesting behavior here: starting with $\mu = 0.4$, the conductance stays almost the same, i.e., is better than what the other algorithms find. Looking at Figure 11.2c,d which show the sizes of the detected communities, we can see that PageRank-Nibble finds really large communities that, starting with $\mu = 0.5$ contain almost half of the nodes of the whole graph. The explanation for this is that if you randomly cut a graph in two parts equally-sized parts, in expectation you also split the neighbors of every node in two equally-sized parts. Therefore, for every node u in expectation about half of u's neighbors are in the same part as u and the other half is in the other part. This yields a conductance of 0.5. Therefore, if the initial local PageRank approximation visits enough nodes, a large enough prefix of the sorted nodes will have a lower conductance than the actual community. Further, if the actual community is not perfectly detected, its conductance is higher and therefore the effect starts already around $\mu = 0.4$.

The Local T algorithm consistently finds too large communities that also have high conductance values except for some range around $\mu = 0.6$, where probably the community size is just right to get low conductance values regardless of the chosen cut. Only when starting with a clique on graphs with low values of μ, the size and conductance are correct, as expected from the good F_1-scores.

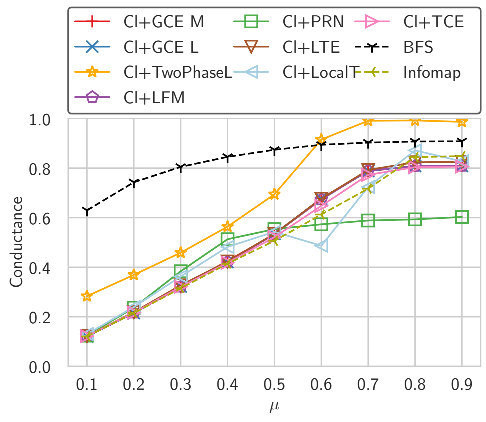
Concerning the sizes, the density-based algorithms GCE M, GCE L and LFM as well as TCE and LTE find approximately the right size, though for larger communities they tend to discover too few nodes. Starting with a clique improves this, the detected community sizes now rather correspond to the correct size which is always returned by the BFS baseline by design. In our previous experiments, LTE tended to detect too large communities for $\mu \geq 0.8$, this was due to a similar bug with the clique initialization as in LocalT.

For Two-Phase L we can clearly see that above $\mu = 0.5$, the algorithm finds smaller and smaller communities on average, meaning that more and more often it decides that the detected community is no community and discards it.

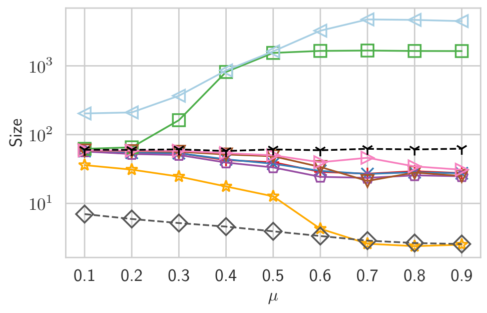
In Figure 11.2e,f we report the average F_1 score of every detected community with the best ground truth community regardless if that community contains the seed node. For



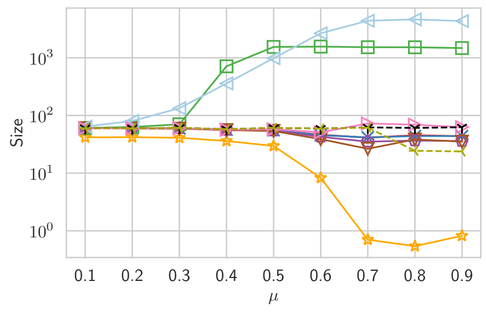
(a) $n = 5\,000$, big



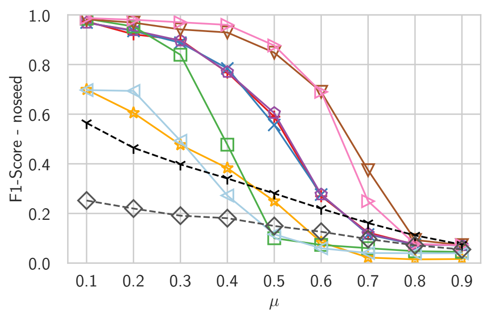
(b) $n = 5\,000$, big



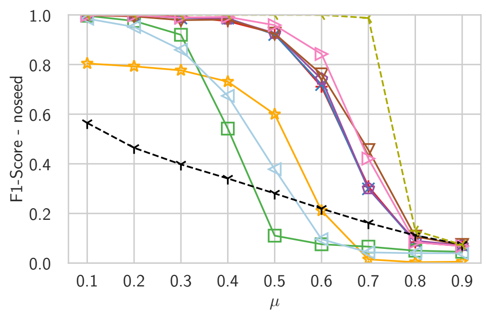
(c) $n = 5\,000$, big



(d) $n = 5\,000$, big



(e) $n = 5\,000$, big



(f) $n = 5\,000$, big

Figure 11.2: Different measures for the 5000 node unweighted graph with disjoint big communities.

GCE M, GCE L and LFM we can clearly see that they get much better scores compared to the comparison considering only the ground truth community of the seed node. This shows that these algorithms find communities in which the seed node is not strongly embedded. For the algorithms starting with a clique, this effect is much weaker, though for higher values of μ it is visible for all algorithms. Existing work frequently ignores this e.g., by just considering the cover that is detected by iteratively applying the algorithm to nodes that do not yet belong to a community [LFK09; Hua+11].

11.4.4 Overlapping Communities

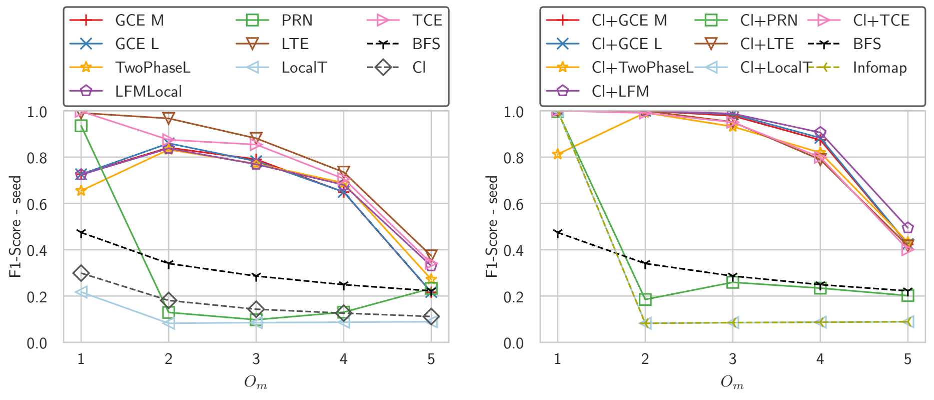
Our parameter set for overlapping communities is inspired by the highly overlapping parameter set used in [Lee+10]. All nodes belong to the same number of communities O_m which is in the range $[1, 5]$. We modified the parameters slightly by not setting the average degree but the minimum degree to $18 \cdot O_m$. This has the effect that the minimum internal degree of a community stays the same with increasing values of O_m. Using NetworKit, we calculated the expected average degree for this minimum degree and used these as parameters for the LFR generator.

Figure 11.3 shows the results for these networks. Note that here we do not evaluate if an algorithm is able to discover all or a specific community of a node, we compare to the community of the seed node that is best matched. The performance of LocalT is extremely poor, it cannot detect any overlapping communities and without a clique as initialization, it cannot even detect the non-overlapping communities. PageRank-Nibble is also unable to discover any communities starting for $O_m \geq 2$, which is not really surprising as the PageRank approximation has no way to decide for one of the communities of the seed node. The other algorithms perform better, LTE being best when starting without a clique. The largest clique of the seed node as initialization again improves the performance, and, quite interestingly, when starting with a clique, the density-based algorithms GCE M, GCE L and LFM perform better than the triangle-based algorithms LTE and TCE. A possible explanation for this could be that triangles are less helpful in the case of overlapping communities as a node can have triangles with neighbors of any community. Infomap completely fails to discover the overlapping community structure, but this is expected as we only used the variant for disjoint communities.

Weighted Graphs

For the weighted graphs, we use the parameter set “Weighted” in Table 11.1, those are the parameters also used by Lancichinetti et al. [LF09b]. This means that we have a single community size range, $[20, 100]$. We further choose three values for the topological mixing parameter $\mu_t = 0.3$, 0.5, 0.8. The weight mixing parameter is varied from 0.1 to 0.9 in each of these sets. For each μ_w we generate 20 instances and run the algorithms from 20 randomly chosen seed nodes.

For the weighted graphs, we do not compare LocalT as it only works on unweighted graphs and it is not clear how to generalize it to weighted graphs, the authors discuss this as future work. In Figure 11.4 we show the results for the other algorithms. For each



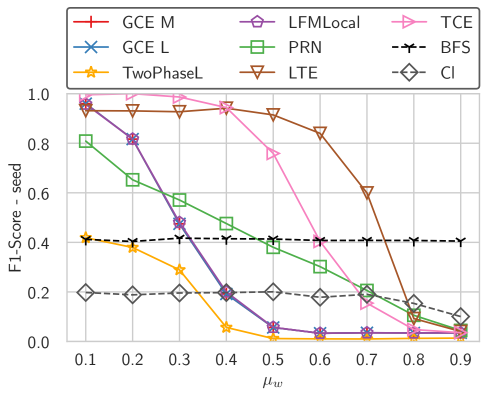
(a) Plain algorithms (b) Clique-based algorithms and Infomap

Figure 11.3: Results for overlapping communities with LFR graphs with parameter set "Overlapping" as specified in Table 11.1.

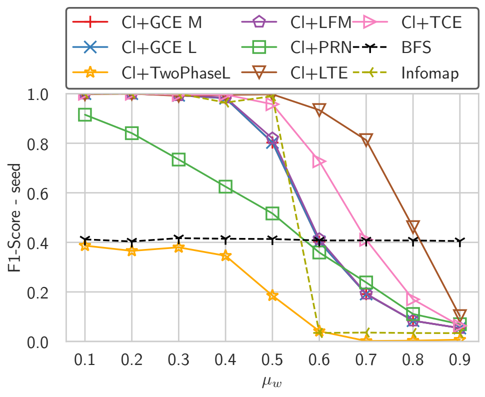
value of the topological mixing parameter μ_t we plot the weight mixing parameter μ_w on the x-axis and the resulting average F_1 score on the y-axis.

Comparing the two plots at $\mu_t = 0.3$ and $\mu_t = 0.5$, we can see that while most algorithms even improve their performance at $\mu_t = 0.5$, LTE has a worse performance. A possible explanation for this is that its score of the community is based on the presence of triangles. While TCE is also based on triangles for the decision which node to add next, its score of the community is solely based on the density and should thus be independent of the structure of the graph as long as the weights are correctly distributed. This can be seen by its performance which is even better for $\mu_t = 0.5$. When initialized with a clique, LTE has fewer problems though and even outperforms Infomap for high values of μ_w. This is probably due to its community score being based on the structure of the graph and thus taking weights less into account. Two-Phase L performs similarly well or worse than our baseline BFS, apparently the two-phase process is not so suited for this kind of weighted graph. The performance of the other density-based algorithms GCE M, GCE L and LFM is similar to their performance on unweighted graphs.

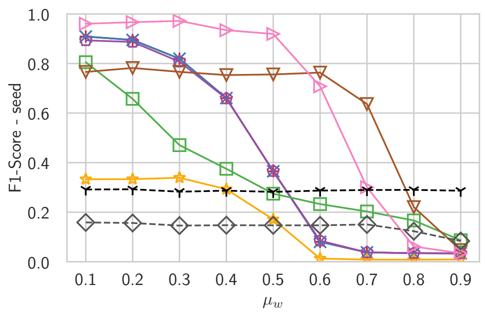
For $\mu_t = 0.8$, none of the algorithms is able to accurately detect the community structure anymore. This is most likely due to the low intra-community degree. Note that the chosen values of $k = 20$ and $k_{\max} = 50$ result in a minimum degree of 10, which means that at $\mu_t = 0.8$, the low-degree nodes have only two neighbors in their own community. PageRank-Nibble seems to be least affected by this, though its performance is also significantly worse than for $\mu_t < 0.8$. LTE performs worse than the other algorithms as it is most affected by the lack of triangles. Cliques also only lead to small improvements, this is most likely also due to the low internal degree. In the previous version of these experiments, due to a bug, no improvements were visible for cliques at all, as the code did not consider the edge weights from the seed node to the clique when determining the maximum-weight clique. The still good performance of Infomap shows, though, that in



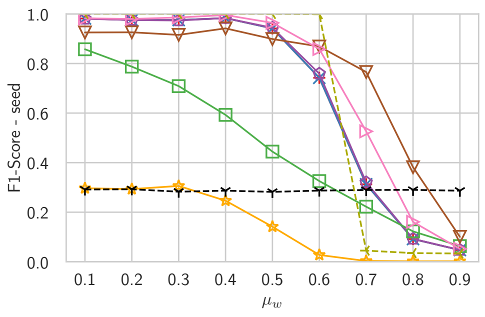
(a) $\mu_t = 0.3$, F_1-score



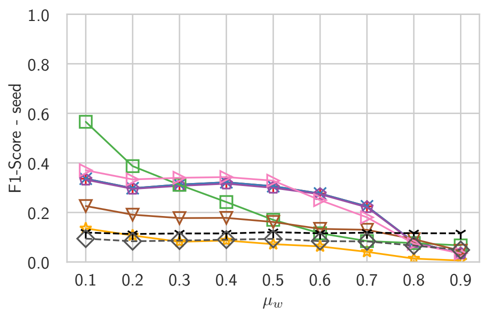
(b) $\mu_t = 0.3$, F_1-score



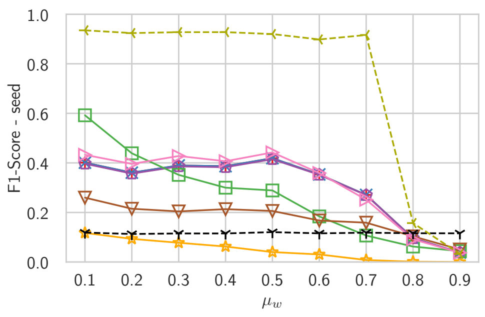
(c) $\mu_t = 0.5$, F_1-score



(d) $\mu_t = 0.5$, F_1-score



(e) $\mu_t = 0.8$, F_1-score



(f) $\mu_t = 0.8$, F_1-score

Figure 11.4: Avg. F_1-scores on the LFR benchmark with the parameter set Weighted, as specified in Table 11.1.

principle the community structure is still detectable.

11.4.5 Facebook Graphs

The synthetic graphs of benchmarks, like LFR, are an excellent way to compare algorithms and can show their quality. However, as discussed in Chapter 2, good performance on synthetic graphs does not necessarily translate to good performance on real world graphs. Good performance on real world graphs is also an important quality of algorithms. After all, we design our algorithms with the goal in mind to be able to find communities on real graphs. Therefore, we test and evaluate the algorithms here on real world social networks, more specifically the Facebook 100 data set that we have discussed already in Section 2.2.3. It consists of 100 Facebook social networks from 100 universities and colleges from September 2005.

In our evaluation, we compare against the community structure induced by the dormitories. These are not ground truth communities, though. Therefore, our results only show how well the communities detected by the algorithms and the attribute correlate. This has many flaws, one of them being that the attribute is missing for some nodes. Another is that the subgraph induced by one of the attribute values is not necessarily connected, in preliminary tests we found that they often even consist of more than two connected components. As these flaws affect all algorithms equally, we still consider this a valid evaluation.

We pick 100 random nodes as seed nodes on each graph and execute the algorithms for each of them. When picking the seed nodes, we ensure that each of them has the dormitory attribute set. Figure 11.5 shows the resulting F_1-scores for all 100 graphs, which we sorted after the F_1-score of LTE.

Clearly, LTE is the best-performing algorithm, in particular when starting with a single seed node. There, on most graphs, none of the other algorithms even exceeds the performance of the baseline BFS. When starting from the maximum clique, TCE, GCE M, GCE L and LFM perform much better and now better than the BFS baseline on many graphs but still worse than LTE. LTE also profits from starting from a clique, but only slightly. PageRank-Nibble does not seem to profit from the clique at all, it appears rather that results are becoming worse when starting from a clique. The Two-Phase L algorithm often decides that no community was discovered, on some of the graphs it even decided that for all 100 seed nodes. While Infomap is among the better algorithms, it does not outperform LTE on most of the graphs.

11.4.6 Running Times

In Table 11.2 we present the running times (wall time) and detected community sizes of the local community detection algorithms averaged over all 100 Facebook graphs and over the overlapping LFR benchmark graphs. While they show different rankings, one can still find some similarities. First of all, LocalT is the slowest algorithm by a large margin and returns the largest communities. LTE is second-slowest, followed by TCE on the LFR graphs. A main difference between the Facebook and the LFR graphs is the time required

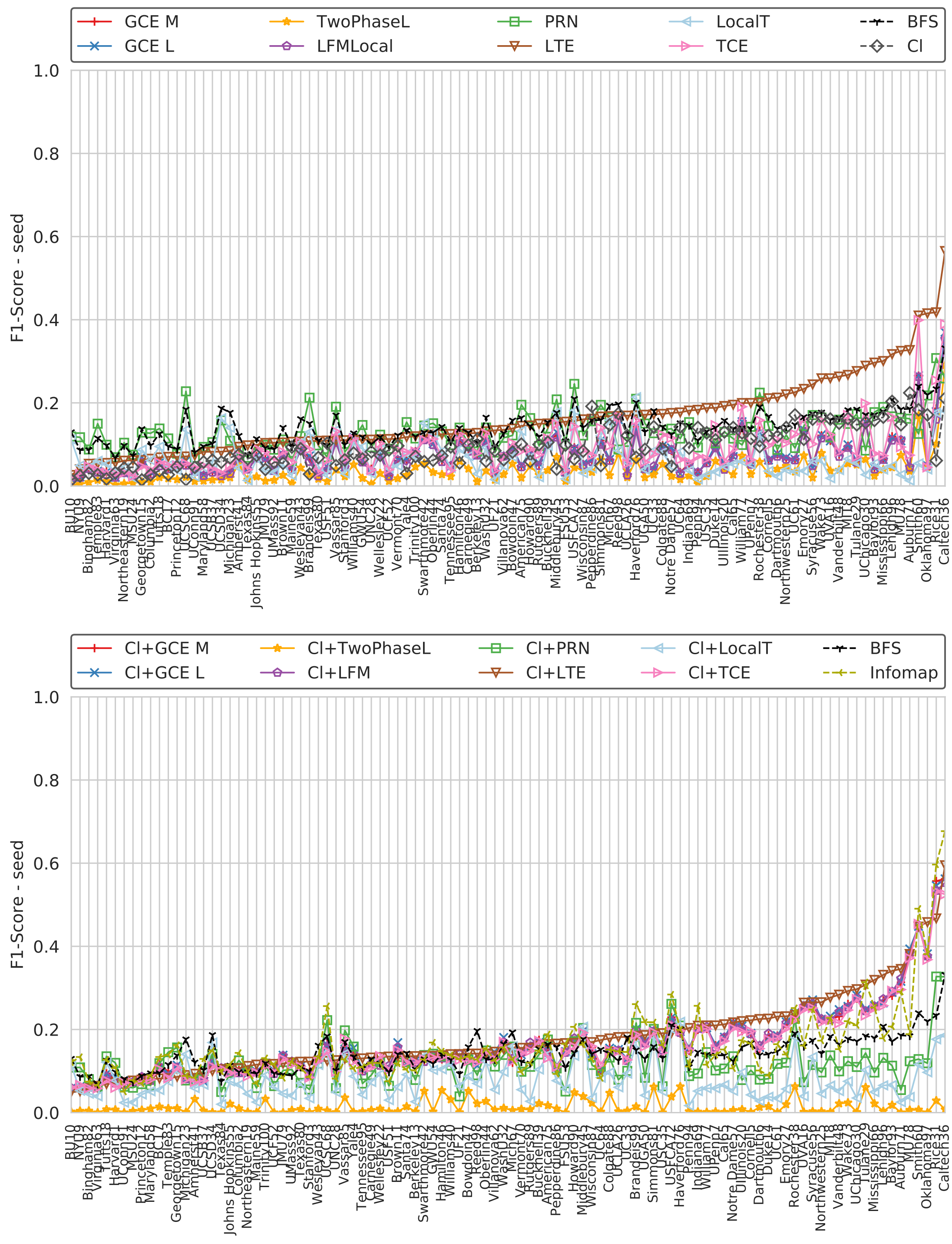


Figure 11.5: Average F_1-scores of the algorithms on all 100 Facebook graphs. The scores are calculated by treating the dormitory attribute as communities. The graphs are sorted by the F_1-score of LTE.

Table 11.2: Average times (ms) per seed node and average community sizes of the algorithms.

(a) Overlapping LFR Benchmark.			**(b) Facebook Networks.**		
	Size	**Time (ms)**		**Size**	**Time (ms)**
Cl	7	0.4	TwoPhaseL	46	1.8
Cl+PRN	420	1.7	PRN	976	3.7
PRN	607	2.5	GCE M	292	6.6
Cl+GCE M	322	3.6	GCE L	287	8.2
GCE M	420	4.1	TCE	313	11.3
Cl+GCE L	315	4.3	LFMLocal	240	12.5
Cl+TwoPhaseL	233	5.0	Cl	14	16.8
GCE L	419	5.2	Cl+TwoPhaseL	45	20.9
LFMLocal	277	6.0	Cl+PRN	1079	21.5
Cl+LFM	266	6.0	Cl+TCE	924	50.3
TwoPhaseL	337	7.0	Cl+GCE M	1012	51.8
TCE	347	11.2	Cl+GCE L	924	54.5
Cl+TCE	327	11.2	Cl+LFM	914	87.3
LTE	311	29.0	LTE	263	106.5
Cl+LTE	303	29.1	Cl+LTE	378	150.8
LocalT	1616	56.6	LocalT	8063	1063.6
Cl+LocalT	1616	57.3	Cl+LocalT	8333	1098.5

for listing all maximal cliques. While it is really fast on the LFR graphs, it is slower than executing any local community detection algorithm apart from LTE and LocalT on the Facebook networks. A possible explanation could be the different structure of the Facebook networks with larger cliques and higher degrees, though the average clique size on the Facebook networks is with 14 only twice as large as the average on the LFR graphs, which is 7. PageRank-Nibble is quite fast on both types of networks, closely followed by the simple expansion strategies GCE M and GCE L. LFM with the more advanced removal strategy is slightly slower. The Two-Phase L algorithm is the fastest algorithm on the Facebook networks, but there it also hardly finds communities so probably it does not only find communities without the seed node as we saw in earlier experiments but the expansion also stops early on these networks. On the LFR benchmark where the performance was similar to the simpler density-based algorithms, Two-Phase L is slightly slower than these algorithms but still faster than the triangle-based algorithms.

11.4.7 Summary

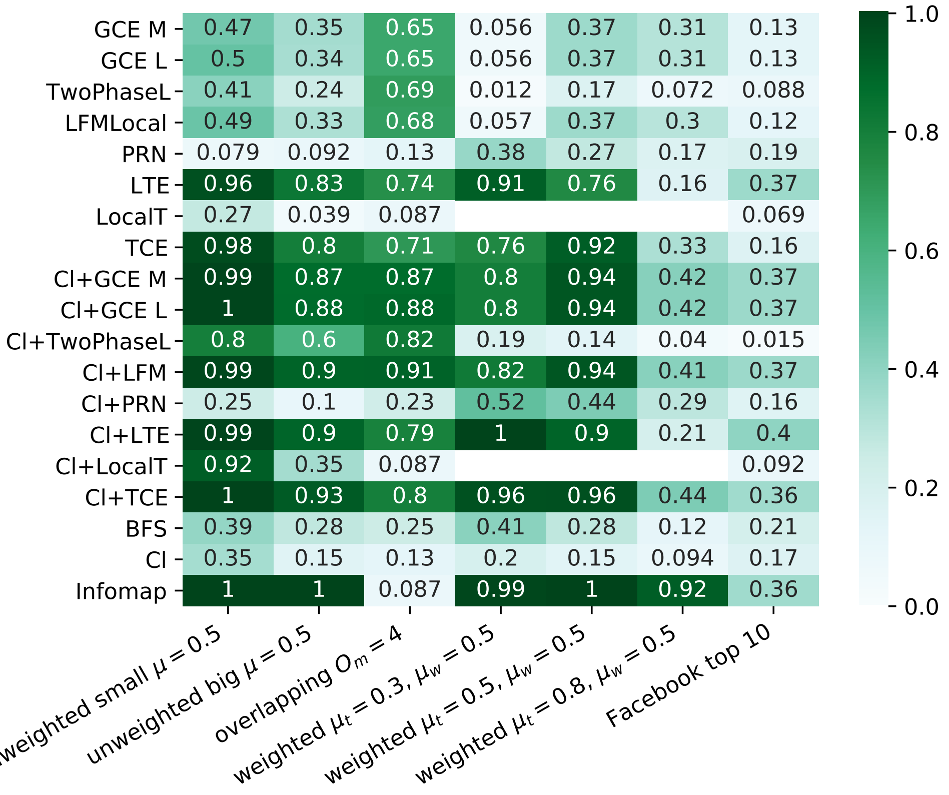
In Figure 11.6 we summarize the experimental results concerning the quality of the found communities. For each type of LFR graph with non-overlapping communities, we show the result for $\mu = 0.5$ as this is for all graphs a value where many algorithms still find reasonable results but differences between algorithms are already visible. For LFR graphs with overlapping communities we picked 4 communities per node for the same reason. For the set of 100 Facebook graphs we report the average over the 10 graphs where LTE starting with a clique (the best-performing algorithm according to our results) found the best-matching communities. On all considered types of graphs, the algorithms not starting with a clique perform worse than those starting with a clique, though LTE and TCE still show reasonable results for many graphs. The algorithms TwoPhaseL, PageRank-Nibble and LocalT perform worse than the other algorithms on all types of graphs. On unweighted LFR graphs with small communities, all other algorithms find the ground truth (almost) perfectly when starting with a clique. With big communities, TCE starting with a clique wins by a small margin. Note that in our previous experiments where LTE had a bug, LTE starting with a clique found the ground truth even better while now it performs slightly worse. On the Facebook graphs, LTE starting with a clique is the best-performing algorithm. However, with overlapping communities and with edge weights and a not so clear structure (i.e., $\mu_t \geq 0.5$), it seems its triangle-based score is more easily confused and outperformed by the simpler density-based algorithms like LFM or GCE with clique initialization. Our newly proposed TCE algorithm is between them, on weighted LFR graphs it is even among the best-performing algorithms. Compared to the global, non-overlapping Infomap algorithm the results are quite competitive, only on the weighted graphs Infomap performs significantly better. On the Facebook networks, the performance of Infomap is even inferior to some local algorithms. For the overlapping LFR networks, Infomap correctly finds that there is no good non-overlapping community structure.

Overall, these results show that while using a clique as initialization is beneficial on all types of graphs, the choice of the expansion algorithm should always depend on the type of the graph. For weighted graphs, it can be dangerous to depend too much on the structure of the graph as LTE does. For LFR graphs with a highly overlapping community structure, methods solely based on the density outperform those based on triangles which is a bit surprising as one would still expect that intra-cluster edges are structurally more embedded. This could mean that possibly scores based on triangles are not as sophisticated as those based on density and should be further improved.

11.5 Conclusion

We have compared a set of 8 local community detection algorithms, one of them—Triangle Based Community Expansion—being a novel algorithm. For the evaluation we used both synthetically generated benchmark graphs and real-world social networks. We have shown that triangle-based algorithms are superior to solely density-based algorithms, in particular on real-world networks. Further, we have shown that all algorithms benefit from not starting from a single seed node but the maximum clique in the neighborhood of the seed node.

Algorithms based on triangles could also be used to improve global overlapping community detection algorithms such as GCE [Lee+10]. In the case of global algorithms, the



	unweighted small $\mu = 0.5$	unweighted big $\mu = 0.5$	overlapping $O_m = 4$	weighted $\mu_t = 0.3, \mu_w = 0.5$	weighted $\mu_t = 0.5, \mu_w = 0.5$	weighted $\mu_t = 0.8, \mu_w = 0.5$	Facebook top 10
GCE M	0.47	0.35	0.65	0.056	0.37	0.31	0.13
GCE L	0.5	0.34	0.65	0.056	0.37	0.31	0.13
TwoPhaseL	0.41	0.24	0.69	0.012	0.17	0.072	0.088
LFMLocal	0.49	0.33	0.68	0.057	0.37	0.3	0.12
PRN	0.079	0.092	0.13	0.38	0.27	0.17	0.19
LTE	0.96	0.83	0.74	0.91	0.76	0.16	0.37
LocalT	0.27	0.039	0.087				0.069
TCE	0.98	0.8	0.71	0.76	0.92	0.33	0.16
Cl+GCE M	0.99	0.87	0.87	0.8	0.94	0.42	0.37
Cl+GCE L	1	0.88	0.88	0.8	0.94	0.42	0.37
Cl+TwoPhaseL	0.8	0.6	0.82	0.19	0.14	0.04	0.015
Cl+LFM	0.99	0.9	0.91	0.82	0.94	0.41	0.37
Cl+PRN	0.25	0.1	0.23	0.52	0.44	0.29	0.16
Cl+LTE	0.99	0.9	0.79	1	0.9	0.21	0.4
Cl+LocalT	0.92	0.35	0.087				0.092
Cl+TCE	1	0.93	0.8	0.96	0.96	0.44	0.36
BFS	0.39	0.28	0.25	0.41	0.28	0.12	0.21
Cl	0.35	0.15	0.13	0.2	0.15	0.094	0.17
Infomap	1	1	0.087	0.99	1	0.92	0.36

Figure 11.6: Summary of F1-Score - seed values of all types of networks considered. The results for the Facebook networks are averages over the 10 networks where "Cl+LTE" had the best scores. LocalT does not support edge weights and is therefore omitted for weighted graphs.

pre-computation of edge scores and using more efficient arrays instead of hash tables lead to significant speedups as preliminary experiments indicate. Further, our approach could also be extended to dynamic graphs, possibly using a similar approach to [ZB15]. Data structures for maintaining triangle counts [ES09; LSS12] could thereby be used to update edge and node scores.

Part V

Conclusion

12 Conclusion and Outlook

In this thesis we considered different aspects of scalable community detection. We started with the question of how to evaluate community detection algorithms. With EM-LFR, we contribute an important building block that allows to evaluate community detection algorithms on graphs with tens of billions of edges using the exact model of the established LFR benchmark. Our dynamic benchmark generator for overlapping communities on the other hand proposes a very challenging benchmark for which scalable algorithms need to be developed.

For disjoint, density-based communities we propose a scalable distributed community detection algorithm that optimizes the established measures modularity and map equation. We show that it is faster and offers better quality than a state-of-the-art algorithm. Further, we compared different methods to filter edges with respect to the preservation of community structure. Our results show that while some methods are effective at selecting intra-community edges, community detection algorithms find different communities on the filtered graphs. On the other hand, simple random edge selection maintains the structure up to about 50% of the edges on the tested graphs.

For the quasi-threshold editing problem, that has been proposed as a possible community detection algorithm, we developed the first scalable editing heuristic that allows editing graphs with millions of nodes and edges. To allow an evaluation of such heuristics with respect to their quality as well as to allow a more detailed evaluation of the quasi-threshold editing approach, we further developed and evaluated exact algorithms. Due to the high complexity of the problem, they are unable to find solutions for graphs with hundreds of nodes. For some small social networks, we analyze the found solutions and show that there can be thousands of them.

In the last part, we considered the problem of detecting a community around a seed node. Here, our focus was on the quality of the detected communities and a detailed experimental comparison of different approaches. We show that starting with the largest clique in the neighborhood of the seed node significantly improves the results. Further, approaches that use edge scores similar to those used in our work on sparsification perform better both on synthetic benchmark graphs and on the Facebook networks.

Even after working more than five years on the research presented in this thesis[1], there are still many ways to continue this research. For our distributed community detection algorithm, integrating approaches recently proposed as parts of the Leiden algorithm [TWE19] should help to improve the quality, in particular for modularity. For quasi-threshold editing, we are currently pursuing two directions of research. First, we believe that inclusion-minimal editing, as proposed for cographs [Cre20], should be

[1]I started my work as a doctoral researcher in July 2014.

possible with small modifications that might also further boost the quality of our heuristic. Minimal editing might also be useful as an initialization heuristic. Second, both heuristic and exact approaches could be adapted for editing costs. While a first attempt with an adaptation of our exact FPT-based branch-and-bound algorithm was much inferior to an ILP [Spi19], we are currently investigating new ideas for lower bounds.

Community detection is a vast field and there are many variants. While countless community detection algorithms have already been proposed, there are still areas where we are not aware of any scalable, high-quality algorithms. This in particular concerns overlapping communities. Already the experiments on our dynamic generator in Chapter 4 reveal that there is still a lot of room for improvements. In a master's thesis I supervised, we considered further extending the Ego-Splitting framework [ELL17] as, while it is fast, the quality of the communities is not perfect as our experiments show [Wie19]. While we improved the quality significantly, there are still open questions concerning in particular the scalability of the extensions and the improvements for detecting communities in real-world networks that are hard to measure. If we combine dynamic networks with overlapping communities, the problem becomes even more challenging. As a continuation of our work on local community detection, further local community detection algorithms could be adjusted for dynamic graphs similar to [ZB15].

Apart from distributed algorithms, external memory algorithms are also a way to scale community detection algorithms to larger graphs. Based on existing work on label propagation [ASS15] and a bachelor's thesis I supervised [Pla16], in particular semi-external algorithms seem promising. In this model, we keep $\mathcal{O}(\log(n))$ bits for each node in the internal memory while storing the edges on disk. This should allow community detection on significantly larger graphs without requiring a compute cluster.

Hypergraphs are a generalization of graphs consisting of nodes and hyperedges where a hyperedge connects an arbitrary set of nodes instead of just two nodes. This allows a more direct modeling of certain settings. For example, consider a co-authorship network where nodes are authors that are connected if they co-authored a publication. Here, a hyperedge could directly represent a publication instead of connecting all authors of a publication by a clique. While some approaches for community detection in hypergraphs have already been considered [PF11; Kam+19], further generalizations of quality measures and algorithms could be developed.

All measures and algorithms for detecting communities that we considered in this thesis only use the structure of a graph and ignore additional information like node attributes. Those could for example indicate that two people work at the same company or attended the same school. One approach to consider attributes is to add additional edges to the graph that represent attribute similarity [RFP13]. Other approaches directly combine attributes and graph structure into a single quality measure [YML13; Sán+15]. For the first approach, all of the community detection algorithms we present in this thesis could be used. In a bachelor's thesis I co-supervised [Kra15], we considered such an approach for local community detection where we extended local community detection algorithms to consider attributes. We found that there are significant challenges how to evaluate the quality of such algorithms. Further, this was before our work on local community detection and only based on algorithms that did not perform well in our comparison. Adjusting the proposed algorithms for graphs with node attributes to use a better base algorithm might thus be promising. For the second case, first understanding the community detection problem alone certainly helps later developing scalable methods that combine both. Further, scalable algorithms like our distributed community detection algorithms could be adjusted to optimize a quality measure like the one used in [Sán+15].

Bibliography

[ABL10] Yong-Yeol Ahn, James Bagrow, and Sune Lehmann. “Link communities reveal multiscale complexity in networks.” In: *Nature* 466 (2010), pp. 761–764. DOI: `10.1038/nature09182`.

[ACL06] Reid Andersen, Fan Chung, and Kevin Lang. “Local Graph Partitioning using PageRank Vectors.” In: *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS’06)*. IEEE Computer Society Press, 2006, pp. 475–486. DOI: `10.1109/FOCS.2006.44`.

[AHH19] Omer Angel, Remco van der Hofstad, and Cecilia Holmgren. “Limit laws for self-loops and multiple edges in the configuration model.” In: *Annales de l’Institut Henri Poincaré, Probabilités et Statistiques* 55.3 (2019), pp. 1509–1530. DOI: `10.1214/18-AIHP926`.

[Ala+16] Maksudul Alam, Maleq Khan, Anil Vullikanti, and Madhav V. Marathe. “An Efficient and Scalable Algorithmic Method for Generating Large-Scale Random Graphs.” In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 372–383. DOI: `10.1109/SC.2016.31`.

[AM11] Rodrigo Aldecoa and Ignacio Marín. “Deciphering Network Community Structure by Surprise.” In: *PLoS ONE* 6 (Sept. 2011), e24195. DOI: `10.1371/journal.pone.0024195`.

[AMM05] Vicente Arnau, Sergio Mars, and Ignacio Marín. “Iterative Cluster Analysis of Protein Interaction Data.” In: *Bioinformatics* 21.3 (2005), pp. 364–378. DOI: `10.1093/bioinformatics/bti021`.

[ANK13] Nesreen K. Ahmed, Jennifer Neville, and Ramana Rao Kompella. “Network Sampling: From Static to Streaming Graphs.” In: *ACM Transactions on Knowledge Discovery from Data* 8.2 (2013), 7:1–7:56. DOI: `10.1145/2601438`.

[Ans85] Richard P. Anstee. “An Algorithmic Proof of Tutte’s f-Factor Theorem.” In: *Journal of Algorithms* 6.1 (1985), pp. 112–131. DOI: `10.1016/0196-6774(85)90022-7`.

[AP14] Alessia Amelio and Clara Pizzuti. “Overlapping Community Discovery Methods: A Survey.” In: *Social Networks: Analysis and Case Studies*. Ed. by Şule Gündüz-Öğüdücü and A. Şima Etaner-Uyar. Lecture Notes in Social Networks. Springer, 2014, pp. 105–125. DOI: `10.1007/978-3-7091-1797-2_6`.

[AP15] Alessia Amelio and Clara Pizzuti. “Is Normalized Mutual Information a Fair Measure for Comparing Community Detection Methods?” In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining.* IEEE, 2015, pp. 1584–1585. DOI: 10.1145/2808797.2809344.

[APU09] Sitaram Asur, Srinivasan Parthasarathy, and Duygu Ucar. “An event-based framework for characterizing the evolutionary behavior of interaction graphs.” In: *ACM Transactions on Knowledge Discovery from Data* 3.4 (Dec. 2009), 16:1–16:36. DOI: 10.1145/1631162.1631164.

[Arg95] Lars Arge. “The Buffer Tree: A New Technique for Optimal I/O-Algorithms.” In: *Proceedings of the 4th International Workshop on Algorithms and Data Structures (WADS’95).* Lecture Notes in Computer Science. Springer, 1995, pp. 334–345. DOI: 10.1007/3-540-60220-8_74.

[ARW12] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. “Fast local search for the maximum independent set problem.” In: *Journal of Heuristics* 18.4 (2012), pp. 525–547. DOI: 10.1007/s10732-012-9196-4.

[ASS15] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. “(Semi-)External Algorithms for Graph Partitioning and Clustering.” In: *Proceedings of the 17th Meeting on Algorithm Engineering and Experiments (ALENEX’15).* SIAM, 2015, pp. 33–43. DOI: 10.1137/1.9781611973754.4.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. “The input/output complexity of sorting and related problems.” In: *Communications of the ACM* 31.9 (July 1988), pp. 1116–1127. DOI: 10.1145/48529.48535.

[Bac+06] Lars Backstrom, Dan Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. “Group formation in large social networks: membership, growth, and evolution.” In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM Press, 2006, pp. 44–54. DOI: 10.1145/1150402.1150412.

[Bad+13] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge.* Vol. 588. American Mathematical Society, 2013.

[Bad+14] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. “Benchmarking for graph clustering and partitioning.” In: *Encyclopedia of Social Network Analysis and Mining.* Ed. by Jon Rokne and Reda Alhajj. Springer, 2014, pp. 73–82. DOI: 10.1007/978-1-4614-6170-8_23.

[Bae+17] Seung-Hee Bae, Daniel Halperin, Jevin D. West, Martin Rosvall, and Bill Howe. “Scalable and Efficient Flow-Based Community Detection for Large-Scale Graph Analysis.” In: *ACM Transactions on Knowledge Discovery from Data* 11.3 (2017), 32:1–32:30. DOI: 10.1145/2992785.

[Bag08] James Bagrow. “Evaluating local community methods in networks.” In: *Journal of Statistical Mechanics: Theory and Experiment* (2008), P05001. DOI: `10.1088/1742-5468/2008/05/P05001`.

[Bat+13] Joshua D. Batson, Daniel A. Spielman, Nikhil Srivastava, and Shang-Hua Teng. “Spectral sparsification of graphs: theory and algorithms.” In: *Communications of the ACM* 56.8 (2013), pp. 87–94. DOI: `10.1145/2492007.2492029`.

[BB05] Vladimir Batagelj and Ulrik Brandes. “Efficient Generation of Large Random Networks.” In: *Physical Review E* 036113 (2005), p. 036113. DOI: `10.1103/PhysRevE.71.036113`.

[BB13] Sebastian Böcker and Jan Baumbach. “Cluster Editing.” In: *Proceedings of the 9th Conference on Computability in Europe (CiE'13)*. Vol. 7921. Lecture Notes in Computer Science. Springer, 2013, pp. 33–44. DOI: `10.1007/978-3-642-39053-1_5`.

[BBK11] Sebastian Böcker, Sebastian Briesemeister, and Gunnar W. Klau. “Exact Algorithms for Cluster Editing: Evaluation and Experiments.” In: *Algorithmica* 60.2 (2011), pp. 316–334. DOI: `10.1007/s00453-009-9339-7`.

[BDS09] Andreas Beckmann, Roman Dementiev, and Johannes Singler. “Building a Parallel Pipelined External Memory Algorithm Library.” In: *23rd International Parallel and Distributed Processing Symposium (IPDPS'09)*. IEEE Computer Society, 2009, pp. 1–10. DOI: `10.1109/IPDPS.2009.5161001`.

[BH15] Seung-Hee Bae and Bill Howe. “GossipMap: a distributed community detection algorithm for billion-edge directed graphs.” In: *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2015, 27:1–27:12. DOI: `10.1145/2807591.2807668`.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. “Gephi: An Open Source Software for Exploring and Manipulating Networks.” In: *Proceedings of the Third International AAAI Conference on Weblogs and Social Media (ICWSM 2009)*. AAAI Press, 2009, pp. 361–362. URL: `http://aaai.org/ocs/index.php/ICWSM/09/paper/view/154`.

[BHK15] Sharon Bruckner, Falk Hüffner, and Christian Komusiewicz. “A graph modification approach for finding core-periphery structures in protein interaction networks.” In: *Algorithms for Molecular Biology* 10 (2015). DOI: `10.1186/s13015-015-0043-7`.

[Bhu+14] Hasanuzzaman Bhuiyan, Jiangzhuo Chen, Maleq Khan, and Madhav V. Marathe. “Fast Parallel Algorithms for Edge-Switching to Achieve a Target Visit Rate in Heterogeneous Graphs.” In: *Proceedings of the 43rd International Conference on Parallel Processing ICPP 2014*. IEEE Computer Society, 2014, pp. 60–69. DOI: `10.1109/ICPP.2014.15`.

[Bin+16] Timo Bingmann, Michael Axtmann, Emanuel Jöbstel, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. *Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++*. 2016. arXiv: `1608.05634`.

[BKT07] Tanya Berger-Wolf, David Kempe, and Chayant Tantipathananandth. "A Framework For Community Identification in Dynamic Social Networks." In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2007, pp. 717–726. DOI: `10.1145/1281192.1281269`.

[Blo+08] Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. "Fast unfolding of communities in large networks." In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (2008). DOI: `10.1088/1742-5468/2008/10/P10008`.

[Böc+08] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. "A fixed-parameter approach for Weighted Cluster Editing." In: *Proceedings of the 6th Asia-Pacific Bioinformatics Conference (APBC 2008)*. Vol. 6. 2008, pp. 211–220. DOI: `10.1142/9781848161092_0023`.

[Böc+09] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. "Going weighted: Parameterized algorithms for cluster editing." In: *Theoretical Computer Science* 410.52 (Dec. 2009), pp. 5467–5480. DOI: `10.1016/j.tcs.2009.05.006`.

[Boh15] Felix Bohlmann. "Graphclustern durch Zerstören langer induzierter Pfade." Bachelor's thesis. TU Berlin, 2015. URL: `http://fpt.akt.tu-berlin.de/publications/theses/BA-felix-bohlmann.pdf`.

[Bol+04] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. "UbiCrawler: A Scalable Fully Distributed Web Crawler." In: *Software - Practice and Experience* 34.8 (2004), pp. 711–726. DOI: `10.1002/spe.587`.

[Bol+11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks." In: *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*. ACM Press, 2011, pp. 587–596. DOI: `10.1145/1963405.1963488`.

[Bor+15] Michele Borassi, Pierluigi Crescenzi, Michel Habib, Walter A. Kosters, Andrea Marino, and Frank W. Takes. "Fast diameter and radius BFS-based computation in (weakly connected) real-world graphs: With an application to the six degrees of separation games." In: *Theoretical Computer Science* 586 (2015), pp. 59–80. DOI: `10.1016/j.tcs.2015.02.033`.

[Bra+08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Höfer, Zoran Nikoloski, and Dorothea Wagner. "On Modularity Clustering." In: *IEEE Transactions on Knowledge and Data Engineering* 20.2 (Feb. 2008), pp. 172–188. DOI: `10.1109/TKDE.2007.190689`.

[Bra+15a] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. *Fast Quasi-Threshold Editing.* 2015. arXiv: `1504.07379 [cs.DS]`.

[Bra+15b] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. "Fast Quasi-Threshold Editing." In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15).* Vol. 9294. Lecture Notes in Computer Science. Springer, 2015, pp. 251–262. DOI: `10.1007/978-3-662-48350-3_22`.

[BS11] Charles-Edmond Bichot and Patrick Siarry, eds. *Graph Partitioning.* Wiley, 2011. DOI: `10.1002/9781118601181`.

[BS80] Jon Louis Bentley and James B. Saxe. "Generating Sorted Lists of Random Numbers." In: *ACM Transactions on Mathematical Software* 6.3 (1980), pp. 359–364. DOI: `10.1145/355900.355907`.

[Buz+14] Nazar Buzun, Anton Korshunov, Valeriy Avanesov, Ilya Filonenko, Ilya Kozlov, Denis Turdakov, and Hangkyu Kim. "EgoLP: Fast and Distributed Community Detection in Billion-Node Social Networks." In: *Proceedings of the 2014 IEEE International Conference on Data Mining Workshops.* IEEE Computer Society, 2014, pp. 533–540. DOI: `10.1109/ICDMW.2014.158`.

[BV04] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques." In: *Proceedings of the 13th International Conference on World Wide Web (WWW2004).* ACM Press, 2004, pp. 595–602. DOI: `10.1145/988672.988752`.

[Cai96] Leizhen Cai. "Fixed-parameter tractability of graph modification problems for hereditary properties." In: *Information Processing Letters* 58.4 (May 1996), pp. 171–176. DOI: `10.1016/0020-0190(96)00050-6`.

[Car+18] Corrie Jacobien Carstens, Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. "Parallel and I/O-efficient Randomisation of Massive Networks using Global Curveball Trades." In: *Proceedings of the 26th Annual European Symposium on Algorithms (ESA'18).* Leibniz International Proceedings in Informatics. 2018, 11:1–11:15. DOI: `10.4230/LIPIcs.ESA.2018.11`.

[CBS18] Corrie Jacobien Carstens, Annabell Berger, and Giovanni Strona. "A unifying framework for fast randomization of ecological networks with fixed (node) degrees." In: *MethodsX* 5 (2018), pp. 773–780. DOI: `10.1016/j.mex.2018.06.018`.

[Cel86] Pedro Celis. "Robin Hood Hashing." PhD thesis. Computer Science Department, University of Waterloo, 1986. URL: `https://cs.uwaterloo.ca/research/tr/1986/CS-86-14.pdf`.

[Cha+17] Tanmoy Chakraborty, Ayushi Dalmia, Animesh Mukherjee, and Niloy Ganguly. "Metrics for Community Analysis: A Survey." In: *ACM Computing Surveys* 50.4 (2017). DOI: `10.1145/3091106`.

[Chu08] Frank Pok Man Chu. “A simple linear time certifying LBFS-based algorithm for recognizing trivially perfect graphs and their complements.” In: *Information Processing Letters* 107.1 (June 2008), pp. 7–12. DOI: 10.1016/j.ipl.2007.12.009.

[Chy+14] Kyrylo Chykhradze, Anton Korshunov, Nazar Buzun, Roman Pastukhov, Nikolay Kuzyurin, Denis Turdakov, and Hangkyu Kim. “Distributed Generation of Billion-node Social Graphs with Overlapping Community Structure.” In: *Proceedings of the 5th Workshop on Complex Networks CompleNet 2014*. Vol. 549. Studies in Computational Intelligence. Springer, 2014, pp. 199–208. DOI: 10.1007/978-3-319-05401-8_19.

[CK01] Anne Condon and Richard M. Karp. “Algorithms for Graph Partitioning on the Planted Partition Model.” In: *Random Structures and Algorithms* 18.2 (2001), pp. 116–140. DOI: 10.1002/1098-2418(200103)18:2<116::AID-RSA1001>3.0.CO;2-2.

[CKU13] Yudong Chen, Vikas Kawadia, and Rahul Urgaonkar. *Detecting Overlapping Temporal Community Structure in Time-Evolving Networks.* 2013. arXiv: 1303.7226.

[Cla05] Aaron Clauset. “Finding local community structure in networks.” In: *Physical Review E* 72.2 (Aug. 2005), p. 026132. DOI: 10.1103/PhysRevE.72.026132.

[CN85] Norishige Chiba and Takao Nishizeki. “Arboricity and Subgraph Listing Algorithms.” In: *SIAM Journal on Computing* 14.1 (Feb. 1985), pp. 210–223. DOI: 10.1137/0214017.

[Cop+19] Lauranne Coppens, Jonathan De Venter, Sandra Mitrovic, and Jochen De Weerdt. “A comparative study of community detection techniques for large evolving graphs.” In: *LEG@ ECML: The third International Workshop on Advances in Managing and Mining Large Evolving Graphs collocated with ECML-PKDD.* Springer. 2019. URL: https://leg-ecmlpkdd19.loria.fr/articles/LEGECML-PKDD_2019_paper_5.pdf.

[CR19] Rémy Cazabet and Giulio Rossetti. “Challenges in Community Discovery on Temporal Networks.” In: *Temporal Network Theory.* Ed. by Petter Holme and Jari Saramäki. Computational Social Sciences. Springer, 2019, pp. 181–197. DOI: 10.1007/978-3-030-23495-9_10.

[Cre20] Christophe Crespelle. “Linear-Time Minimal Cograph Editing.” 2020. URL: https://perso.ens-lyon.fr/christophe.crespelle/publications/SUB_minimal-cograph-editing.pdf.

[CS11] Jie Chen and Ilya Safro. “Algebraic distance on graphs.” In: *SIAM Journal on Scientific Computing* 33.6 (2011), pp. 3468–3490. DOI: 10.1137/090775087.

[CZG09] Jiyang Chen, Osmar R. Zaïane, and Randy Goebel. "Local Community Identification in Social Networks." In: *Proceedings of the 2009 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE Computer Society, 2009, pp. 237–242. DOI: 10.1109/ASONAM.2009.14.

[Dam08] Peter Damaschke. "Fixed-Parameter Enumerability of Cluster Editing and Related Problems." In: *Theory of Computing Systems* 46.2 (2008), pp. 261–283. DOI: 10.1007/s00224-008-9130-1.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Communications of the ACM* 51.1 (2008), pp. 107–113. DOI: 10.1145/1327452.1327492.

[DGG13] Maximilien Danisch, Jean-Loup Guillaume, and Bénédicte Le Grand. "Towards multi-ego-centred communities: a node similarity approach." In: *International Journal of Web Based Communities* 9.3 (2013), pp. 299–322. DOI: 10.1504/IJWBC.2013.054906.

[DHC95] Hassan Ali Dawah, Bradford A. Hawkins, and Michael F. Claridge. "Structure of the Parasitoid Communities of Grass-Feeding Chalcid Wasps." In: *Journal of Animal Ecology* 64.6 (1995), pp. 708–720. DOI: 10.2307/5850.

[DKS08] Roman Dementiev, Lutz Kettner, and Peter Sanders. "STXXL: standard template library for XXL data sets." In: *Software - Practice and Experience* 38.6 (Apr. 2008), pp. 589–637. DOI: 10.1002/spe.844.

[DP17] Pål Grønås Drange and Michał Pilipczuk. "A Polynomial Kernel for Trivially Perfect Editing." In: *Algorithmica* 80.12 (Dec. 2017), pp. 3481–3524. DOI: 10.1007/s00453-017-0401-6.

[Dra+15] Pål Grønås Drange, Markus Sortland Dregi, Daniel Lokshtanov, and Blair D. Sullivan. "On the Threshold of Intractability." In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*. Vol. 9294. Lecture Notes in Computer Science. Springer, 2015. DOI: 10.1007/978-3-662-48350-3_35.

[Dua+12] Dongsheng Duan, Yuhua Li, Ruixuan Li, and Zhengding Lu. "Incremental K-clique clustering in dynamic social networks." In: *Artificial Intelligence Review* 38.2 (Aug. 2012), pp. 129–147. DOI: 10.1007/s10462-011-9250-x.

[Ebb+08] Peter Ebbes, Zan Huang, Arvind Rangaswamy, and Hari P. Thadakamalla. "Sampling large-scale social networks: Insights from simulated networks." In: *Proceedings of the 18th Annual Workshop on Information Technologies and Systems (WITS 2008)*. 2008, pp. 49–54. URL: https://pennstate.pure.elsevier.com/en/publications/sampling-large-scale-social-networks-insights-from-simulated-netw.

[EH80] Roger B. Eggleton and Derek Allan Holton. “Simple and multigraphic realizations of degree sequences.” In: *Proceedings of the Eighth Australian Conference on Combinatorial Mathematics*. Lecture Notes in Mathematics. Springer, 1980, pp. 155–172. DOI: `10.1007/BFb0091817`.

[EK10] David Easley and Jon M. Kleinberg. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, July 2010. URL: `https://www.cs.cornell.edu/home/kleinber/networks-book/`.

[ELL17] Alessandro Epasto, Silvio Lattanzi, and Renato Paes Leme. “Ego-Splitting Framework: from Non-Overlapping to Overlapping Clusters.” In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2017, pp. 145–154. DOI: `10.1145/3097983.3098054`.

[ELS10] David Eppstein, Maarten Löffler, and Darren Strash. “Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time.” In: *Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC’10)*. Lecture Notes in Computer Science. Springer, 2010, pp. 403–414. DOI: `10.1007/978-3-642-17517-6_36`.

[ELS13] David Eppstein, Maarten Löffler, and Darren Strash. “Listing All Maximal Cliques in Large Sparse Real-World Graphs.” In: *ACM Journal of Experimental Algorithmics* 18 (Nov. 2013), 3.1:1–3.1:21. DOI: `10.1145/2543629`.

[Emm+16] Scott Emmons, Stephen G. Kobourov, Mike Gallant, and Katy Börner. “Analysis of Network Clustering Algorithms and Cluster Quality Metrics at Scale.” In: *PLoS ONE* 11.7 (July 2016), pp. 1–18. DOI: `10.1371/journal.pone.0159161`.

[Epp+12] David Eppstein, Michael T. Goodrich, Darren Strash, and Lowell Trott. “Extended dynamic subgraph statistics using h-index parameterized data structures.” In: *Theoretical Computer Science* 447 (Aug. 2012), pp. 44–52. DOI: `10.1016/j.tcs.2011.11.034`.

[ER59] Paul Erdős and Alfred Rényi. “On Random Graphs I.” In: *Publicationes Mathematicae Debrecen* 6 (1959), pp. 290–297.

[ES09] David Eppstein and Emma S. Spiro. “The h-Index of a Graph and Its Application to Dynamic Subgraph Statistics.” In: *Algorithms and Data Structures, 11th International Symposium (WADS’09)*. Ed. by Frank Dehne, Marina Gavrilova, Jörg-Rüdiger Sack, and Csaba D. Tóth. Vol. 5664. Lecture Notes in Computer Science. Springer, Aug. 2009, pp. 278–289. DOI: `10.1007/978-3-642-03367-4_25`.

[FA19] Md Abdul Motaleb Faysal and Shaikh Arifuzzaman. “Distributed Community Detection in Large Networks using An Information-Theoretic Approach.” In: *2019 IEEE International Conference on Big Data*. IEEE Computer Society, 2019, pp. 4773–4782. DOI: `10.1109/BigData47090.2019.9005562`.

[Fan+14] Meng Fanrong, Zhu Mu, Zhou Yong, and Zhou Ranran. "Local Community Detection in Complex Networks Based on Maximum Cliques Extension." In: *Mathematical Problems in Engineering* 2014 (2014). Article ID 653670, 12 pages. DOI: 10.1155/2014/653670.

[FB07] Santo Fortunato and Marc Barthélemy. "Resolution limit in community detection." In: *Proceedings of the National Academy of Science of the United States of America* 104.1 (2007), pp. 36–41. DOI: 10.1073/pnas.0605965104.

[FH16] Santo Fortunato and Darko Hric. "Community detection in networks: A user guide." In: *Physics Reports* 659 (2016), pp. 1–44. DOI: 10.1016/j.physrep.2016.09.002.

[FKW14] Tobias Fleck, Andrea Kappes, and Dorothea Wagner. "Graph Clustering with Surprise: Complexity and Exact Solutions." In: *Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'14)*. Vol. 8327. Lecture Notes in Computer Science. Springer, 2014, pp. 223–234. DOI: 10.1007/978-3-319-04298-5_20.

[Fon+11] Luciano da Fontoura Costa, Osvaldo N. Oliveira Jr., Gonzalo Travieso, Francisco Aparecido Rodrigues, Paulino Ribeiro Villas Boas, Lucas Antiqueira, Matheus Palhares Viana, and Luis Enrique Correa Rocha. "Analyzing and modeling real-world phenomena with complex networks: a survey of applications." In: *Advances in Physics* 60.3 (2011), pp. 329–412. DOI: 10.1080/00018732.2011.572452.

[For10] Santo Fortunato. "Community detection in graphs." In: *Physics Reports* 486.3–5 (2010), pp. 75–174. DOI: 10.1016/j.physrep.2009.11.002.

[Fun19] Park K. Fung. "InfoFlow: A Distributed Algorithm to Detect Communities According to the Map Equation." In: *Big Data and Cognitive Computing* 3.3 (2019), p. 42. DOI: 10.3390/bdcc3030042.

[FY48] Ronald A. Fisher and Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research.* Oliver and Boyd, London, 1948. URL: http://www.worldcat.org/oclc/14222135.

[FZB14] Justin Fagnan, Osmar R. Zaïane, and Denilson Barbosa. "Using triads to identify local community structure in social networks." In: *Proceedings of the 2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining.* IEEE Computer Society, 2014, pp. 108–112. DOI: 10.1109/ASONAM.2014.6921568.

[Gal57] David Gale. "A theorem on flows in networks." In: *Pacific Journal of Mathematics* 7.2 (1957), pp. 1073–1082. DOI: 10.2140/pjm.1957.7.1073.

[GD03] Pablo Gleiser and Leon Danon. "Community Structure in Jazz." In: *Advances in Complex Systems* 6.4 (2003), pp. 565–573. DOI: 10.1142/S0219525903001067.

[GDC10] Derek Greene, Dónal Doyle, and Pádraig Cunningham. “Tracking the Evolution of Communities in Dynamic Social Networks.” In: *Proceedings of the 2010 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE Computer Society, 2010, pp. 176–183. DOI: 10.1109/ASONAM.2010.17.

[Gel19] John Gelhausen. “A Comparative Study of Overlapping Community Detection Algorithms.” Bachelor’s thesis. Department of Informatics, Karlsruhe Institute of Technology (KIT), 2019. URL: https://i11www.iti.kit.edu/_media/teaching/theses/ba-gelhausen-19.pdf.

[Gho+18a] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Assefaw H. Gebremedhin. “Scalable Distributed Memory Community Detection Using Vite.” In: *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference*. IEEE, 2018. DOI: 10.1109/HPEC.2018.8547534.

[Gho+18b] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarrià-Miranda, Arif Khan, and Assefaw H. Gebremedhin. “Distributed Louvain Algorithm for Graph Community Detection.” In: *32nd International Parallel and Distributed Processing Symposium (IPDPS’18)*. IEEE Computer Society, 2018, pp. 885–895. DOI: 10.1109/IPDPS.2018.00098.

[Gil59] Edgar Nelson Gilbert. “Random Graphs.” In: *The Annals of Mathematical Statistics* 30.4 (1959), pp. 1141–1144. DOI: 10.1214/aoms/1177706098.

[GKW14] Robert Görke, Andrea Kappes, and Dorothea Wagner. “Experiments on Density-Constrained Graph Clustering.” In: *ACM Journal of Experimental Algorithmics* 19 (Sept. 2014), 1.6:1.1–1.6:1.31. DOI: 10.1145/2638551.

[GMC10] Benjamin H. Good, Yves-Alexandre de Montjoye, and Aaron Clauset. “Performance of modularity maximization in practical contexts.” In: *Physical Review E* 81.046106 (2010). DOI: 10.1103/PhysRevE.81.046106.

[GMZ03] Christos Gkantsidis, Milena Mihail, and Ellen W. Zegura. “The Markov Chain Simulation Method for Generating Connected Power Law Random Graphs.” In: *Proceedings of the 5th Workshop on Algorithm Engineering and Experiments (ALENEX’03)*. SIAM, 2003, pp. 16–25.

[GN02] Michelle Girvan and Mark E. J. Newman. “Community structure in social and biological networks.” In: *Proceedings of the National Academy of Science of the United States of America* 99.12 (2002), pp. 7821–7826. DOI: 10.1073/pnas.122653799.

[Gör+12] Robert Görke, Roland Kluge, Andrea Schumm, Christian L. Staudt, and Dorothea Wagner. *An Efficient Generator for Clustered Dynamic Random Networks*. Tech. rep. Karlsruhe Reports in Informatics 2012,17. ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2012. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000029825.

[Got+20a] Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. *Engineering Exact Quasi-Threshold Editing.* 2020. arXiv: 2003.14317 [cs.DS].

[Got+20b] Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. "Engineering Exact Quasi-Threshold Editing." In: *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20).* Vol. 160. Leibniz International Proceedings in Informatics. 2020, 10:1–10:14. DOI: 10.4230/LIPIcs.SEA.2020.10.

[Gra+15] Clara Granell, Richard K. Darst, Alex Arenas, Santo Fortunato, and Sergio Gomez. "Benchmark model to assess community structure in evolving networks." In: *Physical Review E* 92.1 (July 2015). DOI: 10.1103/PhysRevE.92.012805.

[GS18] Catherine Greenhill and Matteo Sfragara. "The switch Markov chain for sampling irregular graphs and digraphs." In: *Theoretical Computer Science* 719 (2018), pp. 1–20. DOI: 10.1016/j.tcs.2017.11.010.

[Gur20] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual.* 2020. URL: http://www.gurobi.com.

[GW89] Martin Grötschel and Yoshiko Wakabayashi. "A cutting plane algorithm for a clustering problem." In: *Mathematical Programming* 45.1-3 (1989), pp. 59–96. DOI: 10.1007/BF01589097.

[HA85] Lawrence Hubert and Phipps Arabie. "Comparing partitions." In: *Journal of Classification* 2.1 (Dec. 1985), pp. 193–218. DOI: 10.1007/BF01908075.

[Hak62] S. Louis Hakimi. "On Realizability of a Set of Integers as Degrees of the Vertices of a Linear Graph. I." In: *Journal of the Society for Industrial and Applied Mathematics* 10.3 (July 1962), pp. 496–506. DOI: 10.1137/0110037.

[Ham+16] Michael Hamann, Gerd Lindner, Henning Meyerhenke, Christian L. Staudt, and Dorothea Wagner. "Structure-preserving sparsification methods for social networks." In: *Social Network Analysis and Mining* 6.1 (2016), 22:1–22:22. DOI: 10.1007/s13278-016-0332-2.

[Ham+17] Michael Hamann, Ulrich Meyer, Manuel Penschuck, and Dorothea Wagner. "I/O-efficient Generation of Massive Graphs Following the LFR Benchmark." In: *Proceedings of the 19th Meeting on Algorithm Engineering and Experiments (ALENEX'17).* SIAM, 2017, pp. 58–72. DOI: 10.1137/1.9781611974768.5.

[Ham+18a] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. "Distributed Graph Clustering Using Modularity and Map Equation." In: *Proceedings of the 24th International Conference on Parallel Processing (Euro-Par 2018).* Vol. 11014. Lecture Notes in Computer Science. Springer, 2018, pp. 688–702. DOI: 10.1007/978-3-319-96983-1_49.

[Ham+18b] Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. "I/O-Efficient Generation of Massive Graphs Following the *LFR* Benchmark." In: *ACM Journal of Experimental Algorithmics* 23 (2018), 2.5:1–2.5:33. DOI: `10.1145/3230743`.

[Har+14] Steve Harenberg, Gonzalo Bello, L. Gjeltema, Stephen Ranshous, Jitendra Harlalka, Ramona Seay, Kanchana Padmanabhan, and Nagiza Samatova. "Community detection in large-scale networks: a survey and empirical evaluation." In: *Wiley Interdisciplinary Reviews: Computational Statistics* 6.6 (2014), pp. 426–439. DOI: `10.1002/wics.1319`.

[Hav55] Vaclav Havel. "A Remark on the Existence of Finite Graphs (in Czech)." In: *Československá Akademie Věd. Časopis Pro Pěstování Matematiky* 80 (1955), pp. 477–480.

[HDF14] Darko Hric, Richard K. Darst, and Santo Fortunato. "Community detection in networks: Structural communities versus ground truth." In: *Physical Review E* 90.6 (2014), p. 062805. DOI: `10.1103/PhysRevE.90.062805`.

[Hel+15] Marc Hellmuth, Nicolas Wieseke, Marcus Lechner, Hans-Peter Lenhof, Martin Middendorf, and Peter F. Stadler. "Phylogenomics with paralogs." In: *Proceedings of the National Academy of Science of the United States of America* 112.7 (2015), pp. 2058–2063. DOI: `10.1073/pnas.1412770112`.

[HH15] Sepp Hartung and Holger H. Hoos. "Programming by Optimisation Meets Parameterised Algorithmics: A Case Study for Cluster Editing." In: *Proceedings of the 9th International Conference on Learning and Intelligent Optimization.* Lecture Notes in Computer Science. Springer, 2015, pp. 43–58. DOI: `10.1007/978-3-319-19084-6_5`.

[HKW16] Tanja Hartmann, Andrea Kappes, and Dorothea Wagner. "Clustering Evolving Networks." In: *Algorithm Engineering - Selected Results and Surveys.* Ed. by Lasse Kliemann and Peter Sanders. Vol. 9220. Lecture Notes in Computer Science. Springer, 2016, pp. 280–329. DOI: `10.1007/978-3-319-49487-6_9`.

[HRW17] Michael Hamann, Eike Röhrs, and Dorothea Wagner. "Local Community Detection Based on Small Cliques." In: *Algorithms* 10.3 (2017), p. 90. DOI: `10.3390/a10030090`.

[HS17] Tobias Heuer and Sebastian Schlag. "Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure." In: *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17).* Ed. by Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman. Vol. 75. Leibniz International Proceedings in Informatics. 2017. DOI: `10.4230/LIPIcs.SEA.2017.21`.

[HTC14] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. "I/O-Efficient Algorithms on Triangle Listing and Counting." In: *ACM Transactions on Database Systems* 39.4 (2014), 27:1–27:30. DOI: `10.1145/2691190.2691193`.

[Hua+11] Jianbin Huang, Heli Sun, Yaguang Liu, Qinbao Song, and Tim Weninger. "Towards Online Multiresolution Community Detection in Large-Scale Networks." In: *PLoS ONE* 6.8 (Aug. 2011), pp. 1–11. DOI: `10.1371/journal.pone.0023829`.

[Jeb+18] Malek Jebabli, Hocine Cherifi, Chantal Cherifi, and Atef Hamouda. "Community detection algorithm evaluation with ground-truth data." In: *Physica A: Statistical Mechanics and its Applications* 492 (2018), pp. 651–706. DOI: `10.1016/j.physa.2017.10.018`.

[Jia+14] Songwei Jia, Lin Gao, Yong Gao, and Haiyang Wang. "Anti-triangle centrality-based community detection in complex networks." In: *IET Systems Biology* 8.3 (2014), pp. 116–125. DOI: `10.1049/iet-syb.2013.0039`.

[Jia+15] Songwei Jia, Lin Gao, Yong Gao, James Nastos, Yijie Wang, Xindong Zhang, and Haiyang Wang. "Defining and identifying cograph communities in complex networks." In: *New Journal of Physics* 17.1 (2015), p. 013044. DOI: `10.1088/1367-2630/17/1/013044`.

[JS16] Emmanuel John and Ilya Safro. "Single- and multi-level network sparsification by algebraic distance." In: *Journal of Complex Networks* 5.3 (Oct. 2016), pp. 352–388. DOI: `10.1093/comnet/cnw025`.

[Kai08] Marcus Kaiser. "Mean clustering coefficients: the role of isolated nodes and leafs on clustering measures for small-world networks." In: *New Journal of Physics* 10.8 (2008). DOI: `10.1088/1367-2630/10/8/083042`.

[Kam+19] Bogumił Kamiński, Valérie Poulin, Paweł Prałat, Przemysław Szufel, and François Théberge. "Clustering via hypergraph modularity." In: *PLoS ONE* 14.11 (2019), pp. 1–15. DOI: `10.1371/journal.pone.0224307`.

[KK98] George Karypis and Vipin Kumar. "A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering." In: *Journal of Parallel and Distributed Computing* 48 (Jan. 1998), pp. 71–95. DOI: `10.1006/jpdc.1997.1403`.

[KN11] Brian Karrer and Mark E. J. Newman. "Stochastic blockmodels and community structure in networks." In: *Physical Review E* 83.1 (2011), p. 016107. DOI: `10.1103/PhysRevE.83.016107`.

[Knu93] Donald E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing.* Addison-Wesley, 1993.

[Kol+14] Tamara G. Kolda, Ali Pinar, Todd D. Plantenga, and C. Seshadhri. "A Scalable Generative Graph Model with Community Structure." In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C424–C452. DOI: `10.1137/130914218`.

[KR15] Tatsuro Kawamoto and Martin Rosvall. "Estimating the resolution limit of the map equation in community detection." In: *Physical Review E* 91 (2015), p. 012809. DOI: `10.1103/PhysRevE.91.012809`.

[Kra15] Jonas Krautter. "Iterative Local Community Detection Combining Graph Structure and Attribute Similarity." Bachelor's thesis. Department of Informatics, Karlsruhe Institute of Technology (KIT), 2015. URL: `https://i11www.iti.kit.edu/_media/teaching/theses/ba-krautter-15.pdf`.

[Kri+10] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. "Hyperbolic geometry of complex networks." In: *Physical Review E* 82.3 (2010), p. 036106. DOI: `10.1103/PhysRevE.82.036106`.

[KTV99] Ravi Kannan, Prasad Tetali, and Santosh Vempala. "Simple Markov-chain algorithms for generating bipartite graphs and tournaments." In: *Random Structures and Algorithms* 14.4 (1999), pp. 293–308. DOI: `10.1002/(SICI)1098-2418(199907)14:4<293::AID-RSA1>3.0.CO;2-G`.

[Kum+07] Jussi M. Kumpula, Jari Saramäki, Kimmo Kaski, and János Kertész. "Limited resolution in complex network community detection with Potts model approach." In: *The European Physical Journal B* 56.1 (Mar. 2007), pp. 41–45. DOI: `10.1140/epjb/e2007-00088-4`.

[Kva87] Tarald O. Kvalseth. "Entropy and Correlation: Some Comments." In: *IEEE Transactions on Systems, Man and Cybernetics* 17.3 (1987), pp. 517–519. DOI: `10.1109/TSMC.1987.4309069`.

[KW73] Daniel J. Kleitman and D. L. Wang. "Algorithms for constructing graphs and digraphs with given valences and factors." In: *Discrete Mathematics* 6.1 (1973), pp. 79–88. DOI: `10.1016/0012-365X(73)90037-X`.

[KWZ15] P. R. Kumar, Martin J. Wainwright, and Riccardo Zecchina. *Mathematical Foundations of Complex Networked Information Systems.* Vol. 2141. Lecture Notes in Mathematics. Springer, 2015. DOI: `10.1007/978-3-319-16967-5`.

[LA99] Bjornar Larsen and Chinatsu Aone. "Fast and effective text mining using linear-time document clustering." In: *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and Data Mining.* ACM Press, 1999, pp. 16–22. DOI: `10.1145/312129.312186`.

[Lan+10] Andrea Lancichinetti, Mikko Kivelä, Jari Saramäki, and Santo Fortunato. "Characterizing the Community Structure of Complex Networks." In: *PLoS ONE* 5.8 (Aug. 2010). DOI: `10.1371/journal.pone.0011976`.

[Lan+11] Andrea Lancichinetti, Filippo Radicchi, José J. Ramasco, and Santo Fortunato. "Finding Statistically Significant Communities in Networks." In: *PLoS ONE* 6.4 (Apr. 2011), pp. 1–18. DOI: `10.1371/journal.pone.0018961`.

[LC13] Conrad Lee and Pádraig Cunningham. *Benchmarking community detection methods on social media data.* 2013. arXiv: `1302.0739`.

[Lee+10] Conrad Lee, Fergal Reid, Aaron McDaid, and Neil Hurley. *Detecting highly overlapping community structure by greedy clique expansion.* 2010. arXiv: `1002.1827`.

[LF06] Jure Leskovec and Christos Faloutsos. "Sampling from large graphs." In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2006, pp. 631–636. DOI: 10.1145/1150402.1150479.

[LF09a] Andrea Lancichinetti and Santo Fortunato. "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities." In: *Physical Review E* 80.1 (2009), p. 016118. DOI: 10.1103/PhysRevE.80.016118.

[LF09b] Andrea Lancichinetti and Santo Fortunato. "Community detection algorithms: A comparative analysis." In: *Physical Review E* 80.5 (Nov. 2009). DOI: 10.1103/PhysRevE.80.056117.

[LFK09] Andrea Lancichinetti, Santo Fortunato, and János Kertész. "Detecting the overlapping and hierarchical community structure of complex networks." In: *New Journal of Physics* 11.033015 (2009). URL: http://www.iop.org/EJ/njp.

[LFR08] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. "Benchmark graphs for testing community detection algorithms." In: *Physical Review E* 78.4 (Oct. 2008), p. 046110. DOI: 10.1103/PhysRevE.78.046110.

[Li+15] Yixuan Li, Kun He, David Bindel, and John E. Hopcroft. "Uncovering the Small Community Structure in Large Networks: A Local Spectral Approach." In: *Proceedings of the 24th International Conference on World Wide Web (WWW 2015)*. ACM Press, 2015, pp. 658–668. DOI: 10.1145/2736277.2741676.

[Lin+15] Gerd Lindner, Christian L. Staudt, Michael Hamann, Henning Meyerhenke, and Dorothea Wagner. "Structure-Preserving Sparsification of Social Networks." In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2015. DOI: 10.1145/2808797.2809313.

[Lin+16] Xiao Ling, Jiahai Yang, Dan Wang, Jinfeng Chen, and Liyao Li. "Fast Community Detection in Large Weighted Networks Using GraphX in the Cloud." In: *18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems*. IEEE, 2016, pp. 1–8. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0012.

[Liu+15] Yunlong Liu, Jianxin Wang, Jie You, Jianer Chen, and Yixin Cao. "Edge deletion problems: Branching facilitated by modular decomposition." In: *Theoretical Computer Science* 573 (2015), pp. 63–70. DOI: 10.1016/j.tcs.2015.01.049.

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. June 2014. URL: http://snap.stanford.edu/data.

[LKF05] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. “Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations.” In: *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM Press, 2005, pp. 177–187. DOI: 10.1145/1081870.1081893.

[LSS12] Min Chih Lin, Francisco J. Soulignac, and Jayme Luiz Szwarcfiter. “Arboricity, h-index, and dynamic algorithms.” In: *Theoretical Computer Science* 426 (2012), pp. 75–90. DOI: 10.1016/j.tcs.2011.12.006.

[Lus+04] David Lusseau, Karsten Schneider, Oliver Boisseau, Patti Haase, Elisabeth Slooten, and Steve Dawson. “The Bottlenose Dolphin Community of Doubtful Sound Features a Large Proportion of Long-Lasting Associations.” In: *Behavioral Ecology and Sociobiology* 54.4 (Sept. 2004), pp. 396–405. DOI: 10.1007/s00265-003-0651-y.

[LWP08] Feng Luo, James Zijun Wang, and Eric Promislow. “Exploring local community structures in large networks.” In: *Web Intelligence and Agent Systems: An International Journal* 6.4 (2008), pp. 387–400. DOI: 10.3233/WIA-2008-0147.

[Ma+14] Lianhang Ma, Hao Huang, Qinming He, Kevin Chiew, and Zhenguang Liu. “Toward seed-insensitive solutions to local community detection.” In: *Journal of Intelligent Information Systems* 43.1 (2014), pp. 183–203. DOI: 10.1007/s10844-014-0315-6.

[MC18] Alessandro Muscoloni and Carlo Vittorio Cannistraci. “A nonuniform popularity-similarity optimization (nPSO) model to efficiently generate realistic complex networks with communities.” In: *New Journal of Physics* 20 (2018), p. 052002. DOI: 10.1088/1367-2630/aac06f.

[Meu+13] Wouter Meulemans, Nathalie Henry Riche, Bettina Speckmann, Basak Alper, and Tim Dwyer. “KelpFusion: A Hybrid Set Visualization Technique.” In: *IEEE Transactions on Visualization and Computer Graphics* 19.11 (2013), pp. 1846–1858. DOI: 10.1109/TVCG.2013.76.

[MGH11] Aaron McDaid, Derek Greene, and Neil Hurley. *Normalized Mutual Information to evaluate overlapping community finding algorithms.* 2011. arXiv: arXiv:1110.2515 [physics.soc-ph].

[MH10] Aaron McDaid and Neil Hurley. “Detecting Highly Overlapping Communities with Model-Based Overlapping Seed Expansion.” In: *Proceedings of the 2010 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining.* IEEE Computer Society, 2010, pp. 112–119. DOI: 10.1109/ASONAM.2010.77.

[Mil+03] Ron Milo, Nadav Kashtan, Shalev Itzkovitz, Mark E. J. Newman, and Uri Alon. *On the uniform generation of random graphs with prescribed degree sequences.* 2003. arXiv: cond-mat/0312028.

[ML12] Julian J. Mcauley and Jure Leskovec. “Learning to Discover Social Circles in Ego Networks.” In: *Advances in Neural Information Processing Systems 25 (NIPS 2012)*. 2012, pp. 548–556. URL: `http://papers.nips.cc/paper/4532-learning-to-discover-social-circles-in-ego-networks`.

[MP16] Ulrich Meyer and Manuel Penschuck. “Generating Massive Scale-Free Networks under Resource Constraints.” In: *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX'16)*. SIAM, 2016, pp. 39–52. DOI: `10.1137/1.9781611974317.4`.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

[MSS03] Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, eds. *Algorithms for Memory Hierarchies, Advanced Lectures*. Vol. 2625. Lecture Notes in Computer Science. Springer, 2003. DOI: `10.1007/3-540-36574-5`.

[MSS14] Henning Meyerhenke, Peter Sanders, and Christian Schulz. “Partitioning of Complex Networks via Size-constrained Clustering.” In: *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Ed. by Joachim Gudmundsson and Jyrki Katajainen. Vol. 8504. Lecture Notes in Computer Science. Springer, 2014, pp. 351–363. DOI: `10.1007/978-3-319-07959-2_30`.

[MZ03] Anil Maheshwari and Norbert Zeh. “A Survey of Techniques for Designing I/O-Efficient Algorithms.” In: *Algorithms for Memory Hierarchies, Advanced Lectures*. Ed. by Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn. Vol. 2625. Lecture Notes in Computer Science. Springer, 2003, pp. 36–61. DOI: `10.1007/3-540-36574-5_3`.

[Nas15] James Nastos. “Utilizing graph classes for community detection in social and complex networks.” PhD thesis. University of British Columbia, 2015. DOI: `10.14288/1.0074429`.

[New10] Mark E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010.

[NG04] Mark E. J. Newman and Michelle Girvan. “Finding and evaluating community structure in networks.” In: *Physical Review E* 69.026113 (2004), pp. 1–16. DOI: `10.1103/PhysRevE.69.026113`.

[NG10] James Nastos and Yong Gao. “A Novel Branching Strategy for Parameterized Graph Modification Problems.” In: *Proceedings of the 4th International Conference on Combinatorial Optimization and Applications*. Vol. 2. Lecture Notes in Computer Science. Springer, 2010, pp. 332–346. DOI: `10.1007/978-3-642-17461-2_27`.

[NG13] James Nastos and Yong Gao. “Familial groups in social networks.” In: *Social Networks* 35.3 (July 2013), pp. 439–450. DOI: `10.1016/j.socnet.2013.05.001`.

[Ngu+11] Nam P. Nguyen, Thang N. Dinh, Sindhura Tokala, and My T. Thai. “Overlapping communities in dynamic networks: their detection and mobile applications.” In: *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking (MOBICOM 2011)*. ACM Press, 2011, pp. 85–96. DOI: `10.1145/2030613.2030624`.

[Nic+13] Bobo Nick, Conrad Lee, Pádraig Cunningham, and Ulrik Brandes. “Simmelian backbones: amplifying hidden homophily in Facebook networks.” In: *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE Computer Society, 2013, pp. 525–532. DOI: `10.1145/2492517.2492569`.

[NOB14] Arlind Nocaj, Mark Ortmann, and Ulrik Brandes. “Untangling hairballs - from 3 to 14 degrees of separation.” In: *Proceedings of the 22nd International Symposium on Graph Drawing (GD’14)*. Ed. by Christian A. Duncan and Antonios Symvonis. Vol. 8871. Lecture Notes in Computer Science. Springer, 2014, pp. 101–112. DOI: `10.1007/978-3-662-45803-7_9`.

[NOB15] Arlind Nocaj, Mark Ortmann, and Ulrik Brandes. “Untangling the Hairballs of Multi-Centered, Small-World Online Social Media Networks.” In: *Journal of Graph Algorithms and Applications* 19.2 (2015), pp. 595–618. DOI: `10.7155/jgaa.00370`.

[NTV12] Blaise Ngonmang, Maurice Tchuente, and Emmanuel Viennet. “Local Community Identification in Social Networks.” In: *Parallel Processing Letters* 22.1 (2012), p. 1240004. DOI: `10.1142/S012962641240004X`.

[OB14] Mark Ortmann and Ulrik Brandes. “Triangle Listing Algorithms: Back from the Diversion.” In: *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX’14)*. Ed. by Catherine C. McGeoch and Ulrich Meyer. SIAM, 2014, pp. 1–8. DOI: `10.1137/1.9781611973198.1`.

[Pag03] Rasmus Pagh. “Basic External Memory Data Structures.” In: *Algorithms for Memory Hierarchies, Advanced Lectures*. Ed. by Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn. Vol. 2625. Lecture Notes in Computer Science. Springer, 2003, pp. 14–35. DOI: `10.1007/3-540-36574-5_2`.

[PBV07] Gergely Palla, Albert-László Barabási, and Tamás Vicsek. “Quantifying social group evolution.” In: *Nature* 446 (Apr. 2007), pp. 664–667. DOI: `10.1038/nature05670`.

[Pei15] Tiago P. Peixoto. “Model Selection and Hypothesis Testing for Large-Scale Network Models with Overlapping Groups.” In: *Physical Review X* 5.1 (2015), p. 011033. DOI: `10.1103/PhysRevX.5.011033`.

[Pen+20] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Scalable Network Generation*. 2020. arXiv: `2003.00736 [cs.DS]`.

[PF11] Li Pu and Boi Faltings. "Hypergraph Clustering for Better Network Traffic Inspection." In: *Proceedings of the 3rd Workshop on Intelligent Security at IJCAI*. 2011, pp. 18–25. URL: https://infoscience.epfl.ch/record/197784.

[Pla16] Oliver Plate. "Semi-externes Clustern von Graphen mit der Louvain-Methode." Bachelor's thesis. Department of Informatics, Karlsruhe Institute of Technology (KIT), 2016. URL: https://i11www.iti.kit.edu/_media/teaching/theses/ba-plate-16.pdf.

[PLC17] Leto Peel, Daniel B. Larremore, and Aaron Clauset. "The Ground Truth About Metadata and Community Detection in Networks." In: *Science Advances* 3.5 (2017), e1602548. DOI: 10.1126/sciadv.1602548.

[PPF15] Costas Panagiotakis, Harris Papadakis, and Paraskevi Fragopoulou. "Local Community Detection via Flow Propagation." In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2015, pp. 81–88. DOI: 10.1145/2808797.2808892.

[PSZ10] Dhruv Parthasarathy, Devavrat Shah, and Tauhid Zaman. *Leaders, Followers, and Community Detection*. 2010. arXiv: 1011.0774 [stat.ML].

[Que+13] Xinyu Que, Fabio Checconi, Fabrizio Petrini, Teng Wang, and Weikuan Yu. *Lightning-fast community detection in social media: A scalable implementation of the louvain algorithm*. Tech. rep. Tech. Rep. AU-CSSE-PASL/13-TR01. Department of Computer Science and Software Engineering, Auburn University, 2013.

[Que+15] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A. Gunnels. "Scalable Community Detection with the Louvain Algorithm." In: *29th International Parallel and Distributed Processing Symposium (IPDPS'15)*. IEEE Computer Society, 2015, pp. 28–37. DOI: 10.1109/IPDPS.2015.59.

[RAB09] Martin Rosvall, Daniel Axelsson, and Carl T. Bergstrom. "The map equation." In: *The European Physical Journal Special Topics* 178.1 (2009), pp. 13–23. DOI: 10.1140/epjst/e2010-01179-1.

[Rad+04] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. "Defining and identifying communities in networks." In: *Proceedings of the National Academy of Science of the United States of America* 101.9 (2004), pp. 2658–2663. DOI: 10.1073/pnas.0400054101.

[Rah+07] Sven Rahmann, Tobias Wittkop, Jan Baumbach, Marcel Martin, Anke Truß, and Sebastian Böcker. "Exact and Heuristic Algorithms for Weighted Cluster Editing." In: *Proceedings of the 6th Annual International Conference on Computational Systems Bioinformatics (CSB 2007)*. Vol. 6. 2007, pp. 391–401. DOI: 10.1142/9781860948732_0040.

[RB06] Jörg Reichardt and Stefan Bornholdt. "Statistical Mechanics of Community Detection." In: *Physical Review E* 74.016110 (2006), pp. 1–16. DOI: 10.1103/PhysRevE.74.016110.

[RFP13] Yiye Ruan, David Fuhry, and Srinivasan Parthasarathy. "Efficient Community Detection in Large Networks Using Content and Links." In: *Proceedings of the 22nd International Conference on World Wide Web (WWW'13)*. ACM Press, 2013, pp. 1089–1098. DOI: `10.1145/2488388.2488483`.

[RN11] Randolf Rotta and Andreas Noack. "Multilevel local search algorithms for modularity clustering." In: *ACM Journal of Experimental Algorithmics* 16 (July 2011), 2.3:2.1–2.3:2.27. DOI: `10.1145/1963190.1970376`.

[RPS15] Jaideep Ray, Ali Pinar, and C. Seshadhri. "A stopping criterion for Markov chains when generating independent random graphs." In: *Journal of Complex Networks* 3.2 (2015), pp. 204–220. DOI: `10.1093/comnet/cnu041`.

[Rys57] Herbert J. Ryser. "Combinatorial Properties of Matrices of Zeros and Ones." In: *Canadian Journal of Mathematics* 9 (1957), pp. 371–377. DOI: `10.4153/CJM-1957-044-3`.

[SAA09] Paolo Simonetto, David Auber, and Daniel Archambault. "Fully Automatic Visualisation of Overlapping Sets." In: *Computer Graphics Forum* 28.3 (2009), pp. 967–974. DOI: `10.1111/j.1467-8659.2009.01452.x`.

[Sán+15] Patricia Iglesias Sánchez, Emmanuel Müller, Uwe Leo Korn, Klemens Böhm, Andrea Kappes, Tanja Hartmann, and Dorothea Wagner. "Efficient Algorithms for a Robust Modularity-Driven Clustering of Attributed Graphs." In: *Proceedings of the 2015 SIAM International Conference on Data Mining*. SIAM, 2015, pp. 100–108. DOI: `10.1137/1.9781611974010.12`.

[San00] Peter Sanders. "Fast Priority Queues for Cached Memory." In: *ACM Journal of Experimental Algorithmics* 5 (2000), p. 7. DOI: `10.1145/351827.384249`.

[San98] Peter Sanders. "Random Permutations on Distributed, External and Hierarchical Memory." In: *Information Processing Letters* 67.6 (1998), pp. 305–309. DOI: `10.1016/S0020-0190(98)00127-6`.

[ŠB11] Lovro Šubelj and Marko Bajec. "Unfolding communities in large complex networks: Combining defensive and offensive label propagation for core extraction." In: *Physical Review E* 83.3 (2011), p. 036103. DOI: `10.1103/PhysRevE.83.036103`.

[SBV09] M. Ángeles Serrano, Marián Boguñá, and Alessandro Vespignani. "Extracting the multiscale backbone of complex weighted networks." In: *Proceedings of the National Academy of Science of the United States of America* 106.16 (2009), pp. 6483–6488. DOI: `10.1073/pnas.0808904106`.

[Sch07] Satu Elisa Schaeffer. "Graph Clustering." In: *Computer Science Review* 1.1 (Aug. 2007), pp. 27–64. DOI: `10.1016/j.cosrev.2007.05.001`.

[SHW17] Neha Sengupta, Michael Hamann, and Dorothea Wagner. "Benchmark Generator for Dynamic Overlapping Communities in Networks." In: *Proceedings of the 2017 IEEE International Conference on Data Mining*. IEEE Computer Society, 2017, pp. 415–424. DOI: `10.1109/ICDM.2017.51`.

[SHZ15] Wolfgang E. Schlauch, Emőke Ágnes Horvát, and Katharina A. Zweig. “Different flavors of randomness: comparing random graph models with fixed degree sequences.” In: *Social Network Analysis and Mining* 5.1 (2015), 36:1–36:14. DOI: 10.1007/s13278-015-0267-z.

[Slo+19] George M. Slota, Jonathan W. Berry, Simon D. Hammond, Stephen L. Olivier, Cynthia A. Phillips, and Sivasankaran Rajamanickam. “Scalable Generation of Graphs for Benchmarking HPC Community-Detection Algorithms.” In: *SC’19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Press, 2019. DOI: 10.1145/3295500.3356206.

[SM16] Christian L. Staudt and Henning Meyerhenke. “Engineering Parallel Algorithms for Community Detection in Massive Networks.” In: *IEEE Transactions on Parallel and Distributed Systems* 27.1 (2016), pp. 171–184. DOI: 10.1109/TPDS.2015.2390633.

[SMM14] Christian L. Staudt, Yassine Marrakchi, and Henning Meyerhenke. “Detecting communities around seed nodes in complex networks.” In: *IEEE International Conference on Big Data*. Ed. by Jimmy J. Lin, Jian Pei, Xiaohua Hu, Chang Wo, Raghunath Nambiar, Charu Aggarwal, Nick Cercone, Vasant Honavar, Jun Huan, Bamshad Mobasher, and Saumyadipta Pyne. IEEE Computer Society, 2014, pp. 62–69. DOI: 10.1109/BigData.2014.7004373.

[SN97] Tom A.B. Snijders and Krzysztof Nowicki. “Estimation and Prediction of Stochastic Blockmodels for Graphs with Latent Block Structure.” In: *Journal of Classification* 14 (1997), pp. 75–100. DOI: 10.1007/s003579900004.

[Spi19] Jonas Spinner. “Weighted F-free Edge Editing.” Bachelor’s thesis. Department of Informatics, Karlsruhe Institute of Technology (KIT), 2019. URL: https://i11www.iti.kit.edu/_media/teaching/theses/ba-spinner-19.pdf.

[SPR11] Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. “Local Graph Sparsification for Scalable Clustering.” In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD’11)*. ACM Press, 2011, pp. 721–732. DOI: 10.1145/1989323.1989399.

[SRD13] Tanwistha Saha, Huzefa Rangwala, and Carlotta Domeniconi. “Sparsification and Sampling of Networks for Collective Classification.” In: *Proceedings of the 6th International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction (SBP 2013)*. Vol. 7812. Lecture Notes in Computer Science. Springer, 2013, pp. 293–302. DOI: 10.1007/978-3-642-37210-0_32.

[SSM16] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. “NetworKit: A tool suite for large-scale complex network analysis.” In: *Network Science* 4.4 (Dec. 2016), pp. 508–530. DOI: 10.1017/nws.2016.20.

[ST15] Julian Shun and Kanat Tangwongsan. "Multicore triangle computations without tuning." In: *Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE 2015*. IEEE Computer Society, 2015, pp. 149–160. DOI: 10.1109/ICDE.2015.7113280.

[SW50] Georg Simmel and Kurt H. Wolff. *The Sociology of Georg Simmel*. Simon and Schuster, 1950.

[TAD15] Vincent A. Traag, Rodrigo Aldecoa, and Jean-Charles Delvenne. "Detecting Communities Using Asymptotical Surprise." In: *Physical Review E* 92.2 (2015), p. 022816. DOI: 10.1103/PhysRevE.92.022816.

[TB11] Chayant Tantipathananandth and Tanya Berger-Wolf. "Finding Communities in Dynamic Social Networks." In: *Proceedings of the 2011 IEEE International Conference on Data Mining*. IEEE Computer Society, 2011, pp. 1236–1241. DOI: 10.1109/ICDM.2011.67.

[TDN11] Vincent A. Traag, P. Van Dooren, and Y. Nesterov. "Narrow scope for resolution-limit-free community detection." In: *Physical Review E* 84.1 (July 2011), p. 016114. DOI: 10.1103/PhysRevE.84.016114.

[The13] The Lemur Project. *ClueWeb12 Web Graph*. http://www.lemurproject.org/clueweb12/webgraph.php. Nov. 2013.

[TKD13] Vincent A. Traag, G. Krings, and P. Van Dooren. "Significant Scales in Community Structure." In: *Nature Scientific Reports* 3.2930 (2013). DOI: 10.1038/srep02930.

[TMP12] Amanda L. Traud, Peter J. Mucha, and Mason A. Porter. "Social structure of Facebook networks." In: *Physica A: Statistical Mechanics and its Applications* 391.16 (Aug. 2012), pp. 4165–4180. DOI: 10.1016/j.physa.2011.12.021.

[TWE19] Vincent A. Traag, Ludo Waltman, and Nees Jan van Eck. "From Louvain to Leiden: guaranteeing well-connected communities." In: *Nature Scientific Reports* 9 (2019), p. 5233. DOI: 10.1038/s41598-019-41695-z.

[Uga+11] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. *The Anatomy of the Facebook Social Graph*. 2011. arXiv: 1111.4503.

[VEB09] Nguyen Xuan Vinh, Julien Epps, and James Bailey. "Information theoretic measures for clusterings comparison: is a correction for chance necessary?" In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM Press, 2009, pp. 1073–1080. DOI: 10.1145/1553374.1553511.

[Vit87] Jeffrey S. Vitter. "An efficient algorithm for sequential random sampling." In: *ACM Transactions on Mathematical Software* 13.1 (1987), pp. 58–67. DOI: 10.1145/23002.23003.

[VL16] Fabien Viger and Matthieu Latapy. "Efficient and simple generation of random simple connected graphs with prescribed degree sequence." In: *Journal of Complex Networks* 4.1 (Mar. 2016). Source code available at `https://www-complexnetworks.lip6.fr/~latapy/FV/generation.html`, pp. 15–37. DOI: `10.1093/comnet/cnv013`.

[VR12] Alcides Viamontes Esquivel and Martin Rosvall. *Comparing network covers using mutual information.* Feb. 2012. arXiv: `1202.0425 [math-ph]`.

[Wic+14] Charith Wickramaarachchi, Marc Frincu, Patrick Small, and Viktor K. Prasanna. "Fast parallel algorithm for unfolding of communities in large graphs." In: *Proceedings of the 2014 IEEE High Performance Extreme Computing Conference.* IEEE, 2014, pp. 1–6. DOI: `10.1109/HPEC.2014.7040973`.

[Wie19] Armin Wiebigke. "Engineering Overlapping Community Detection based on the Ego-Splitting Framework." MA thesis. Department of Informatics, Karlsruhe Institute of Technology (KIT), 2019. URL: `https://i11www.iti.kit.edu/_media/teaching/theses/ma-wiebigke-19.pdf`.

[Wit+11] Tobias Wittkop, Dorothea Emig, Anke Truß, Mario Albrecht, Sebastian Böcker, and Jan Baumbach. "Comprehensive cluster analysis with Transitivity Clustering." In: *Nature Protocols* 6 (2011), pp. 285–295. DOI: `10.1038/nprot.2010.197`.

[Wol65] E. S. Wolk. "A Note on "The Comparability Graph of a Tree"." In: *Proceedings of the American Mathematical Society* 16.1 (Feb. 1965), pp. 17–20. DOI: `10.2307/2033992`.

[WS98] Duncan J. Watts and Steven H. Strogatz. "Collective dynamics of 'small-world' networks." In: *Nature* 393.6684 (June 1998), pp. 440–442. DOI: `10.1038/30918`.

[WW07] Silke Wagner and Dorothea Wagner. *Comparing Clusterings – An Overview.* Tech. rep. 2006-04. Universität Karlsruhe (TH), 2007. URL: `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000011477`.

[Xia+18] Ju Xiang, Hui-Jia Li, Zhan Bu, Zhen Wang, Mei-Hua Bao, Liang Tang, and Jian-Ming Li. "Critical analysis of (Quasi-)Surprise for community detection in complex networks." In: *Nature Scientific Reports* 8 (2018), p. 14459. DOI: `10.1038/s41598-018-32582-0`.

[XKS13] Jierui Xie, Stephen Kelley, and Boleslaw K. Szymanski. "Overlapping community detection in networks: The state-of-the-art and comparative study." In: *ACM Computing Surveys* 45.4 (2013), Article 43, 35 pages. DOI: `10.1145/2501654.2501657`.

[XSL11] Jierui Xie, Boleslaw K. Szymanski, and Xiaoming Liu. "SLPA: Uncovering Overlapping Communities in Social Networks via A Speaker-listener Interaction Dynamic Process." In: *Proceedings of the 11th IEEE International Conference on Data Mining - Workshops*. IEEE Computer Society, 2011, pp. 344–349. DOI: `10.1109/ICDMW.2011.154`.

[YCC96] Jing-Ho Yan, Jer-Jeong Chen, and Gerard J. Chang. "Quasi-threshold graphs." In: *Discrete Applied Mathematics* 69.3 (Aug. 1996), pp. 247–255. DOI: `10.1016/0166-218X(96)00094-7`.

[YL12a] Jaewon Yang and Jure Leskovec. "Community-Affiliation Graph Model for Overlapping Network Community Detection." In: *Proceedings of the 2012 IEEE International Conference on Data Mining*. IEEE Computer Society, 2012, pp. 1170–1175. DOI: `10.1109/ICDM.2012.139`.

[YL12b] Jaewon Yang and Jure Leskovec. "Defining and Evaluating Network Communities Based on Ground-Truth." In: *Proceedings of the 2012 IEEE International Conference on Data Mining*. IEEE Computer Society, 2012, pp. 745–754. DOI: `10.1109/ICDM.2012.138`.

[YL14] Jaewon Yang and Jure Leskovec. "Structure and Overlaps of Ground-Truth Communities in Networks." In: *ACM Transactions on Intelligent Systems and Technology* 5.2 (Apr. 2014), 26:1–26:35. DOI: `10.1145/2594454`.

[YL15] Jaewon Yang and Jure Leskovec. "Defining and evaluating network communities based on ground-truth." In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213. DOI: `10.1007/s10115-013-0693-z`.

[YML13] Jaewon Yang, Julian McAuley, and Jure Leskovec. "Community Detection in Networks with Node Attributes." In: *Proceedings of the 2013 IEEE 13th International Conference on Data Mining*. IEEE, 2013, pp. 1151–1156. DOI: `10.1109/ICDM.2013.167`.

[Zac77] Wayne W. Zachary. "An Information Flow Model for Conflict and Fission in Small Groups." In: *Journal of Anthropological Research* 33 (1977), pp. 452–473. DOI: `10.1086/jar.33.4.3629752`.

[Zah+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. "Apache Spark: a unified engine for big data processing." In: *Communications of the ACM* 59.11 (2016), pp. 56–65. DOI: `10.1145/2934664`.

[ZB15] Anita Zakrzewska and David A. Bader. "A Dynamic Algorithm for Local Community Detection in Graphs." In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2015, pp. 559–564. DOI: `10.1145/2808797.2809375`.

[Zei17] Tim Zeitz. “Engineering Distributed Graph Clustering using MapReduce.” MA thesis. Department of Informatics, Karlsruhe Institute of Technology (KIT), 2017. URL: `https://i11www.iti.kit.edu/_media/teaching/theses/ma-zeitz-17.pdf`.

[Zha13] James Y. Zhao. *Expand and Contract: Sampling graphs with given degrees and other combinatorial families.* 2013. arXiv: `1308.6627`.

[ZN99] Xiao Zhou and Takao Nishizeki. “Edge-Coloring and f-Coloring for Various Classes of Graphs.” In: *Journal of Graph Algorithms and Applications* 3.1 (1999), pp. 1–18. DOI: `10.1007/3-540-58325-4_182`.

[ZY16] Jianping Zeng and Hongfeng Yu. “A study of graph partitioning schemes for parallel graph community detection.” In: *Parallel Computing* 58 (2016), pp. 131–139. DOI: `10.1016/j.parco.2016.05.008`.

[ZY18] Jianping Zeng and Hongfeng Yu. “A Distributed Infomap Algorithm for Scalable and High-Quality Community Detection.” In: *Proceedings of the 47th International Conference on Parallel Processing ICPP 2018.* ACM Press, 2018. DOI: `10.1145/3225058.3225137`.

List of Publications

Book Chapters

Ulrik Brandes, Michael Hamann, Mark Ortmann, and Dorothea Wagner. “On the Persistence of Strongly Embedded Ties.” In: *Computational Social Science in the Age of Big Data: Concepts, Methodologies, Tools, and Applications*. Ed. by Cathleen M. Stützer, Martin Welker, and Marc Egger. Vol. 15. Neue Schriften zur Online-Forschung. Herbert von Halem Verlag, 2018, pp. 213–234. URL: `http://www.halem-verlag.de/computational-social-science-in-the-age-of-big-data/`.

Journal Articles

Gregor Betz, Michael Hamann, Tamara Mchedlidze, and Sophie von Schmettow. “Applying argumentation to structure and visualize multi-dimensional opinion spaces.” In: *Argument & Computation* 10.1 (2019), pp. 23–40. DOI: `10.3233/AAC-181004`.

Michael Hamann and Ben Strasser. “Correspondence between Multilevel Graph Partitions and Tree Decompositions.” In: *Algorithms* 12.9 (2019), p. 198. DOI: `10.3390/a12090198`.

Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. “Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies.” In: *Algorithms* 12.9 (2019), p. 196. DOI: `10.3390/a12090196`.

Michael Hamann and Ben Strasser. “Graph Bisection with Pareto Optimization.” In: *ACM Journal of Experimental Algorithmics* 23.1 (2018), 1.2:1–1.2:34. DOI: `10.1145/3173045`.

Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. “I/O-Efficient Generation of Massive Graphs Following the *LFR* Benchmark.” In: *ACM Journal of Experimental Algorithmics* 23 (2018), 2.5:1–2.5:33. DOI: `10.1145/3230743`.

Rudolf Biczok, Peter Bozsoky, Peter Eisenmann, Johannes Ernst, Tobias Ribizel, Fedor Scholz, Axel Trefzer, Florian Weber, Michael Hamann, and Alexandros Stamatakis. “Two C++ libraries for counting trees on a phylogenetic terrace.” In: *Bioinformatics* 34.19 (2018), pp. 3399–3401. DOI: `10.1093/bioinformatics/bty384`.

Christian L. Staudt, Michael Hamann, Alexander Gutfraind, Ilya Safro, and Henning Meyerhenke. “Generating realistic scaled complex networks.” In: *Applied Network Science* 2.1 (2017), p. 36. DOI: `10.1007/s41109-017-0054-z`.

Michael Hamann, Eike Röhrs, and Dorothea Wagner. “Local Community Detection Based on Small Cliques.” In: *Algorithms* 10.3 (2017), p. 90. DOI: `10.3390/a10030090`.

Michael Hamann, Gerd Lindner, Henning Meyerhenke, Christian L. Staudt, and Dorothea Wagner. “Structure-preserving sparsification methods for social networks.” In: *Social Network Analysis and Mining* 6.1 (2016), 22:1–22:22. DOI: 10.1007/s13278-016-0332-2.

Conference Articles

Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. “Advanced Flow-Based Multilevel Hypergraph Partitioning.” In: *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20)*. Vol. 160. Leibniz International Proceedings in Informatics. 2020, 11:1–11:15. DOI: 10.4230/LIPIcs.SEA.2020.11.

Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. “Engineering Exact Quasi-Threshold Editing.” In: *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20)*. Vol. 160. Leibniz International Proceedings in Informatics. 2020, 10:1–10:14. DOI: 10.4230/LIPIcs.SEA.2020.10.

Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. “Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm.” In: *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*. Leibniz International Proceedings in Informatics. 2019, 52:1–52:17. DOI: 10.4230/LIPIcs.ESA.2019.52.

Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. “Distributed Graph Clustering Using Modularity and Map Equation.” In: *Proceedings of the 24th International Conference on Parallel Processing (Euro-Par 2018)*. Vol. 11014. Lecture Notes in Computer Science. Springer, 2018, pp. 688–702. DOI: 10.1007/978-3-319-96983-1_49.

Corrie Jacobien Carstens, Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. “Parallel and I/O-efficient Randomisation of Massive Networks using Global Curveball Trades.” In: *Proceedings of the 26th Annual European Symposium on Algorithms (ESA'18)*. Leibniz International Proceedings in Informatics. 2018, 11:1–11:15. DOI: 10.4230/LIPIcs.ESA.2018.11.

Neha Sengupta, Michael Hamann, and Dorothea Wagner. “Benchmark Generator for Dynamic Overlapping Communities in Networks.” In: *Proceedings of the 2017 IEEE International Conference on Data Mining*. IEEE Computer Society, 2017, pp. 415–424. DOI: 10.1109/ICDM.2017.51.

Michael Hamann, Ulrich Meyer, Manuel Penschuck, and Dorothea Wagner. “I/O-efficient Generation of Massive Graphs Following the LFR Benchmark.” In: *Proceedings of the 19th Meeting on Algorithm Engineering and Experiments (ALENEX'17)*. SIAM, 2017, pp. 58–72. DOI: 10.1137/1.9781611974768.5.

Christian L. Staudt, Michael Hamann, Ilya Safro, Alexander Gutfraind, and Henning Meyerhenke. “Generating Scaled Replicas of Real-World Complex Networks.” In: *Proceedings of the 5th International Workshop on Complex Networks and their Applications (COMPLEX NETWORKS 2016).* Vol. 693. Studies in Computational Intelligence. Springer, 2016, pp. 17–28. DOI: `10.1007/978-3-319-50901-3_2`.

Michael Hamann and Ben Strasser. “Graph Bisection with Pareto-Optimization.” In: *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX’16).* SIAM, 2016, pp. 90–102. DOI: `10.1137/1.9781611974317.8`.

Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. “Fast Quasi-Threshold Editing.” In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA’15).* Vol. 9294. Lecture Notes in Computer Science. Springer, 2015, pp. 251–262. DOI: `10.1007/978-3-662-48350-3_22`.

Gerd Lindner, Christian L. Staudt, Michael Hamann, Henning Meyerhenke, and Dorothea Wagner. “Structure-Preserving Sparsification of Social Networks.” In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining.* IEEE, 2015. DOI: `10.1145/2808797.2809313`.

Michael Hamann, Tanja Hartmann, and Dorothea Wagner. “Complete Hierarchical Cut-Clustering: A Case Study on Expansion and Modularity.” In: *Graph Partitioning and Graph Clustering: Tenth DIMACS Implementation Challenge.* Ed. by David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Vol. 588. DIMACS Book. American Mathematical Society, 2013, pp. 157–170. DOI: `10.1090/conm/588`.

Michael Hamann, Tanja Hartmann, and Dorothea Wagner. “Hierarchies of Predominantly Connected Communities.” In: *Algorithms and Data Structures, 13th International Symposium (WADS’13).* Vol. 8037. Lecture Notes in Computer Science. Springer, 2013, pp. 365–377. DOI: `10.1007/978-3-642-40104-6_32`.

Technical Reports

Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. *Advanced Flow-Based Multilevel Hypergraph Partitioning.* 2020. arXiv: `2003.12110 [cs.DS]`.

Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. *Engineering Exact Quasi-Threshold Editing.* 2020. arXiv: `2003.14317 [cs.DS]`.

Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Scalable Network Generation.* 2020. arXiv: `2003.00736 [cs.DS]`.

Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. *Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm.* 2019. arXiv: `1907.02053 [cs.DS]`.

Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. *Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies.* 2019. arXiv: `1906.11811 [cs.DS]`.

Corrie Jacobien Carstens, Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. *Parallel and I/O-efficient Randomisation of Massive Networks using Global Curveball Trades.* 2018. arXiv: `1804.08487 [cs.DS]`.

Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. *Distributed Graph Clustering using Modularity and Map Equation.* 2017. arXiv: `1710.09605 [cs.DS]`.

Christian L. Staudt, Michael Hamann, Alexander Gutfraind, Ilya Safro, and Henning Meyerhenke. *Generating realistic scaled complex networks.* 2016. arXiv: `1609.02121 [cs.SI]`.

Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. *I/O-Efficient Generation of Massive Graphs Following the LFR Benchmark.* 2016. arXiv: `1604.08738 [cs.DS]`.

Michael Hamann, Gerd Lindner, Henning Meyerhenke, Christian L. Staudt, and Dorothea Wagner. *Structure-Preserving Sparsification Methods for Social Networks.* 2016. arXiv: `1601.00286 [cs.SI]`.

Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. *Fast Quasi-Threshold Editing.* 2015. arXiv: `1504.07379 [cs.DS]`.

Michael Hamann and Ben Strasser. *Graph Bisection with Pareto-Optimization.* 2015. arXiv: `1504.03812 [cs.DS]`.

Gerd Lindner, Christian L. Staudt, Michael Hamann, Henning Meyerhenke, and Dorothea Wagner. *Structure-Preserving Sparsification of Social Networks.* 2015. arXiv: `1505.00564 [cs.SI]`.

Michael Hamann, Tanja Hartmann, and Dorothea Wagner. *Hierarchies of Predominantly Connected Communities.* 2013. arXiv: `1305.0757 [cs.DS]`.