Karlsruhe Institute of Technology

Engineering Overlapping Community Detection based on the Ego-Splitting Framework

Master Thesis of

Armin Wiebigke

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Michael Hamann

Time Period: 28th February 2019 – 27th August 2019

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, August 26, 2019

Abstract

A community is a set of strongly connected nodes in a network, e.g. a group of friends in a social network. Finding community structures in a network is a problem with many practical applications, revealing information about the structure of the network as well as the interaction between its elementary units. In real-world networks, communities often overlap, meaning that nodes may be part of multiple communities. We evaluate the problem of detecting overlapping communities in a network, using the *ego-splitting* method. Ego-splitting is a framework which first searches for the communities of each node in its local neighborhood, the *ego-net*. Each node is then split into multiple *personas* to disentangle overlapping communities, each persona corresponding to one of the locally detected communities. The result is a persona graph with non-overlapping communities on which detecting communities is comparatively easy. The ego-splitting framework requires two non-overlapping community detection algorithms, one to analyze the ego-net and one to detect the communities on the persona graph. The two algorithms can be chosen freely from the extensive set of well-known algorithms for non-overlapping community detection. While this means that the ego-splitting framework is highly flexible, there is a lack of practical evaluation. We present extensive experimental results for various high quality non-overlapping community detection algorithms and analyze their usefulness in the framework. Our experiments show that choosing the right algorithms is essential to detect high quality communities.

We extend the ego-splitting framework by three additional steps to improve the quality of the detected communities. First, we increase the scope of the local analysis by extending the ego-net, i.e. we add additional nodes to the ego-net. In our experiments, we show that the structure of the communities is improved in the extended ego-net, making it easier to detect the correct communities. The presented ego-net extension may also have useful applications in other algorithms that analyze local community structures. Second, we connect all personas of each node in the persona graph to reflect their relation to one another, improving the quality of the communities detected on the persona graph. Third, we clean up the detected communities to make sure that they are well-connected internally. Our experiments show that the clean-up process considerably improves the quality of the communities. In principle, the clean-up process is independent of the ego-splitting framework, making it possible to clean up communities detected by any overlapping community detection algorithm.

Our results show that our improved version of the ego-splitting framework is able to detect communities of high quality in overlapping networks. Moreover, the running time of our algorithm is comparatively low and does not depend on the structure of the network. On synthetic networks, where each node is part of multiple communities (between two and four), our algorithm outperforms the state-of-the-art algorithms OSLOM and MOSES in both quality and running time.

Deutsche Zusammenfassung

Eine Community ist eine Menge von dicht verbundenden Knoten in einem Netzwerk, z.B eine Gruppe von Freunden in einem sozialen Netzwerk. Das Erkennen von Communities in Netzwerken ist ein Problem mit vielen praktischen Anwendungsbereichen, da es Information sowohl über die Struktur des Netzwerk als auch über die Interaktionen zwischen den einzelnen Teilen enthüllen kann. Eine Community ist eine Menge von Knoten die dicht verbunden sind, z.B. ein Gruppe von Freuden in einen sozialen Netzwerk. In realen Netzwerken sind Communities häufig überlappend, das heißt ein Knoten kann Teil von mehreren Communities sein. Wir evaluieren das Problem der Erkennung von überlappenden Communities basierend auf der *Ego-Splitting* Methode. Ego-Splitting ist ein Framework, das zuerst nach den Communities jedes Knoten in seiner lokalen Nachbarschaft, dem *Ego-Net*, sucht. Dann wird jeder Knoten in eine Menge von *Personas* aufgeteilt, um die überlappenden Communities voneinander zu trennen, wobei jeder Persona eine der lokal gefundenen Communities zugeordnet wird. Man erhält einen Persona-Graph, der nur nicht-überlappende Communities enthält, die vergleichsweise einfach zu erkennen sind. Das Ego-Splitting Framework benötigt zwei Algorithmen zur Erkennung von nicht-überlappenden Communities, einen um das Ego-Net zu analysieren und einen um die Communities im Persona-Graph zu erkennen. Die zwei Algorithmen können frei aus der großen Menge an bekannten Algorithmen zur Erkennung von nicht-überlappenden Communities gewählt werden. Zwar ist das Ego-Splitting Framework dadurch sehr flexibel, aber es gibt einen Mangel an praktischen Evaluationen. Wir präsentieren umfangreiche experimentelle Ergebnisse für verschiedene Algorithmen zur Erkennung von nicht-überlappenden Communities, und analysieren deren Brauchbarkeit für das Framework. Unsere Experimente zeigen, dass die Auswahl der korrekten Algorithmen essentiell ist zur Erkennung von Communities mit hoher Qualität.

Wir erweitern das Ego-Splitting Framework um drei zusätzliche Schritte, um die Qualität der gefundenen Communities zu erhöhen. Erstens erhöhen wir die Umfang der lokalen Analyse, indem wir zusätzlich Knoten zum Ego-Net hinzufügen. In unseren Experimenten zeigen wir, dass die Communities in diesem erweiterten Ego-Net eine bessere Struktur besitzen, wodurch es leichter ist, die korrekten Communities zu erkennen. Die vorgestellte Ego-Net Erweiterung könnte auch für andere Algorithmen Verwendung finden, die lokale Communitystrukturen analysieren. Zweitens verbinden wir alle Personas jedes Knotens im Persona-Graph, um deren Beziehung zueinander hervorzuheben. Dadurch wird die Qualität der erkannten Communities im Persona-Graph erhöht. Drittens bereinigen wir die erkannten Communities, um sicherzustellen, dass sie intern dicht verbunden sind. Unsere Experimente zeigen, dass der Bereinigungsprozess die Qualität der Communities deutlich verbessert. Der Bereinungsprozess ist prinzipiell unabhängig vom Ego-Splitting Framework und kann damit benutzt werden, um die erkannten Communities beliebiger Algorithmen zu bereinigen.

Unsere Ergebnisse zeigen, dass unsere verbesserte Version des Ego-Splitting Frameworks Communities von hoher Qualität auf überlappenden Netzwerken erkennen kann. Außerdem ist die Laufzeit unseres Algorithmus verhältnismäßig niedrig und hängt nicht von der Struktur des Netzwerks ab. Auf synthetischen Netzwerken, in denen jeder Knoten Teil von mehreren (zwei bis vier) Communities ist, übertrifft unser Algorithmus die State-of-the-Art-Algorithmen OSLOM und MOSES sowohl hinsichtlich der Qualität als auch hinsichtlich der Laufzeit.

Contents

1 Introduction **1**

2 Preliminaries **5**

3 Related Work **7**
3.1 Non-overlapping Community Detection 7
3.2 Overlapping Community Detection 10

4 The Ego-Splitting Framework **15**

5 Ego-Net Extension **19**
5.1 Number of Edges 21
5.2 Statistical Significance 22

6 Connecting Personas **29**

7 Community Clean-Up **33**
7.1 Single Community Analysis 34
7.2 Merge Communities 36

8 Experimental Setup **39**
8.1 Implementation Details 39
8.2 Algorithms 40
8.3 Graphs 41
8.4 Evaluation Metrics 42
8.4.1 Ego-Net Structure 43
8.4.2 Ego-Net Clustering 44
8.4.3 Cover 46

9 Experimental Results **47**
9.1 Ego-Net Extension 47
9.1.1 EdgesScore 48
9.1.2 Significance 52
9.1.3 Original vs. Extended Ego-Net 59
9.2 Local Clustering Algorithm 62
9.3 Connecting Personas 66
9.4 Global Clustering Algorithm 68
9.5 Community Clean-Up 72
9.6 Comparison with Other OCD Algorithms 76

10 Conclusion **83**

Bibliography **85**

1. Introduction

In practical and theoretical applications, relations between actors can often be modeled as a network. A network consists of a set of actors represented by *nodes*. The actors have some sort of relations, represented by *edges*, each connecting two nodes. A *community* is a strongly connected set of nodes, i.e. each node is well-connected to the other nodes of the community. An intuitive example is the structure in a social network, where each node represents a person. Each connection in the social network represents a relationship between two people, and the detected communities could be social groups, families, etc. [Fel81]. Detecting such communities (also called *modules* or *clusters*) in a network is known as the *community detection problem*.
Detecting and analyzing the community structure of networks is a highly researched topic in the last years. By understanding the structure of a network, valuable information can be extracted. There is a wide range of practical applications for this, e.g. in biological (functional modules in protein interaction networks [PDFV05]), chemical (biochemical pathways in metabolic networks [HHJ03]), technological (viral marketing in recommendation networks [LAH07]), social (groups of mutual acquaintances in online social networks [MMG+07]), and information (web communities on the internet [FLG+00]) systems. However, there is no universally accepted definition of a community. Typically, a community detection algorithm defines a fitness function which is then optimized to find communities. Numerous techniques have been developed for finding disjoint communities, e.g. by using random walks [RB08], modularity maximization [BGLL08], or label propagation [RAK07]. These algorithms assume that the network can be partitioned into densely connected regions, with few connections between these regions. However, real networks are often not structured like that. For example, a person is usually part of multiple social groups, so a node in a social network may be part of multiple communities. Consequently, communities may overlap, i.e. the communities are not disjoint. In fact, many real-world graphs show such characteristics [RMH11]. The problem of finding such overlapping communities is known as *overlapping community detection*.

Many overlapping community detection algorithms define a fitness function and then greedily expand communities by locally optimizing the fitness. For example, these fitness functions can be based on stochastic models [MH10], stochastic significance [LRRF11] or they compare the internal and outgoing connections of the community [LRMH10]. The communities are often initialized with small random communities or fully connected sets of nodes [LRMH10].

Another approach is to start the community detection on a microscopic (local) level and create global communities using the obtained local information [BKA+14, CRGP14]. Detecting local communities is much easier than directly detecting communities on the entire graph [CRGP14, ELM+15]. Often, such a local community detection is based on *ego-nets* (also called ego-networks), each consisting of the subgraph induced over the neighborhood of a single node in the graph. Epasto et al. [ELM+15] propose that the community detection in the ego-net can be reduced to non-overlapping community detection. This works because even if a node is part of many communities, the chance that a neighbor is in multiple of these communities is quite low. Based on these findings, Epasto et al. [ELPL17] introduced the *ego-splitting* framework for overlapping community detection. The idea of the framework is to use the local structures to detect overlapping communities and then "disentangle" them into non-overlapping communities. By using this approach, they are able to reduce the overlapping community detection problem to the simpler non-overlapping community detection problem. The ego-splitting framework works in two steps, a local ego-net analysis and a global community detection. In both steps, a non-overlapping community detection algorithm is used. Such algorithms are also known as *clustering* algorithms. After finding the local communities in all ego-nets, each node is split into multiple copies, called *personas*. Each persona is assigned to one of the locally detected communities, and receives all edges to nodes inside that community, resulting in the *persona graph*. Ideally, the persona graph does not contain any overlapping communities. In the second step, a clustering algorithm is used on the persona graph. To create the overlapping communities on the original graph, each node is assigned the communities of all its personas. The ego-splitting framework uses two clustering algorithms, the local clustering algorithm to analyze the ego-net and the global clustering algorithm to detect the communities on the persona graph. The algorithms can be chosen freely from the extensive amount of existing clustering algorithms, making the framework very flexible.

Contribution

We extend the ego-splitting framework by three additional steps to improve the quality of the community detection. First, we increase the quality of the local analysis by extending the ego-net, i.e. adding additional nodes that strengthen the communities we want to detect. Second, we propose to connect the personas of each node, as they possess a relation which is not reflected in the persona graph. This provides the global clustering algorithm with additional information, which increases the global quality. Third, we propose an additional clean-up step after the creation of the overlapping communities. We use statistical significance to remove weakly connected nodes from the communities and also add strongly connected nodes.

While the ego-splitting framework is flexible, there is a lack of practical evaluation, especially for different clustering algorithms. As testing all possible combinations of local and global algorithms is impractical, we propose metrics to evaluate the quality of the local clustering in isolation. This evaluation of the ego-net is also useful on its own, as ego-net analysis is also an important part of other algorithms. We provide extensive experimental results for the framework using various clustering algorithms that have shown to provide high quality results. For the evaluation, we use various synthetic and real-world graphs. Additionally, we compare our optimized ego-splitting algorithm against state-of-the-art overlapping community detection algorithms.

Our experiments show that the additional steps of the framework improve the quality of the detected communities, both for the ego-net analysis and the global community detection. We show that the choice of clustering algorithms greatly influences the quality of the detected communities. Our optimized ego-splitting algorithm improves the quality

drastically compared to the “basic” ego-splitting framework and outperforms other state-of-the-art algorithms in both quality and running time on many synthetic graphs.

Thesis Outline

This thesis is structured as follows: First, we present related work and introduce the state-of-the-art algorithms for non-overlapping and overlapping community detection. Then we give a detailed description of the ego-splitting framework as it has been proposed by Epasto et al. [ELPL17]. In Chapter 5, we describe multiple strategies to extend the ego-net. Next, we describe how the personas of a node can be connected in the persona graph. Then we present the process to clean up detected communities. In Chapter 8, we present the configurations and graphs used in the experiments. We also present the quality measures that we use to analyze the experiments. Then we present the experimental results and evaluate them. In the end, we summarize our improvements and discuss possible additional modifications to the framework.

2. Preliminaries

A graph G is defined as $G = (V, E)$, where V is a set of n nodes and $E \subseteq V \times V$ is a set of m edges. In this thesis, we will use only undirected and unweighted graphs, unless stated otherwise. In an undirected graph, each edge between two nodes u and v is a set $\{u, v\}$ of size two. For convenience, we will use the notation (u, v) instead. A graph is called *dense* if $m \in \Theta(n^2)$ and called *sparse* if $m \in O(n)$. An alternative way to describe the structure of the graph is by using an $n \times n$ adjacency matrix A. A_{ij} is equal to 1 if there is an edge $e \in E$ that connects the nodes i and j, and it is 0 otherwise. The degree of node u is defined as $k_u = |\{v|(u, v) \in E\}|$.

A *community* (or *cluster*) $c_i \subseteq V$ is a subset of the nodes of the graph. We say that a node u is assigned to the community c_i if $u \in c_i$. A non-overlapping community detection algorithm takes a graph G and outputs a set of disjoint communities $\{c_1, c_2, \ldots, c_k\}, c_i \cap c_j = \emptyset \forall i, j \neq i$. An overlapping community detection (OCD) algorithm takes a graph G and outputs a set of communities $\{c_1, c_2, \ldots, c_k\}$. These communities may overlap, so a node may be assigned to more than one community. To better distinguish between the communities produced by these two types of community detection algorithms, we use the following notation: We refer to non-overlapping community detection algorithms as *clustering* algorithms. A clustering algorithm outputs a *clustering* $D = \{d_1, d_2, \ldots, d_k\}$ of clusters d_i. An overlapping community detection algorithm outputs a *cover* $C = \{c_1, c_2, \ldots, c_k\}$ of communities c_i.

We call a graph *slightly overlapping* if each node is assigned to two or fewer communities, and *highly overlapping* if each node is assigned to more than two communities.

Given a community c, its degree K_c is the sum of the degrees of all nodes in c. K_c can be split into the internal degree K_c^{in} and the external degree K_c^{out}. K_c^{in} is the sum of the internal degrees of the nodes in c, so it is equal to twice the number of internal edges. K_c^{out} is the number of edges that have exactly one end in c, so K_c^{out} is equal to the size of the cut between c and the rest of the graph.

3. Related Work

There are a multitude of community detection algorithms, both for non-overlapping and overlapping communities. We will restrict ourselves to reviewing the current best algorithms for various approaches. We also present related work that focuses on the evaluation of ego-nets.

3.1 Non-overlapping Community Detection

Non-overlapping communities divide the nodes into disjoint subsets. As stated earlier, we refer to these non-overlapping communities as clusters. Non-overlapping community detection is related to the classic partition problem. The goal of a k-partition is to divide the graph into k sub-sets of roughly even size. One of the simplest partition algorithms is the *minimum-cut* algorithm, which divides the graph into a fixed number of subsets and minimizes the cut between these subsets. In community detection however, the number of clusters is not known beforehand, and the size of the clusters in the given graph can vary greatly. This complicates the problem, and requires adaptive algorithms.

Girvan-Newman

A classical example for community detection is the *Girvan-Newman* algorithm (GN) [GN02]. The algorithm is based on the concept of *edge betweenness*. The edge betweenness of an edge e is defined as the number of shortest paths, between all pairs of vertices, that pass along e. The algorithm finds the edge with the highest edge betweenness, which is likely to lie between two clusters, and removes that edge from the graph. The connected components of the remaining graph are the clusters. Edges are removed iteratively until no edges remain. The output of the algorithm is a hierarchical clustering which represents a tree where the root is the set of all nodes and the leaves are the nodes. By cutting the tree at a given height, we get clusters of a given resolution.

Label Propagation

Label Propagation (LP) [RAK07] is often used because of its simplicity and low running time. Each node is initialized with a unique label. In every iteration of the algorithm, each node adopts the label that the maximum number of its neighbors have, with ties broken randomly. After a short time, densely connected groups should reach a consensus on one label. At the end of the algorithm, nodes with the same label form one cluster. There is

no specific metric or fitness function that is optimized. The algorithm terminates when the number of nodes that changed in the last iteration is smaller than a given threshold θ. Alternatively, the algorithm terminates when a maximum number of iterations is reached. While the algorithm is relatively simple, it also has disadvantages. For example, in dense graphs it is possible that all nodes get the same label, resulting in a trivial cluster that contains all nodes.

Absolute Potts Model

Ronhovde and Nussinov introduced the *Absolute Potts Model* (APM) [RN10] which penalizes for missing edges within a cluster. The cluster detection algorithm of the APM [ELPL17] is a modified version of Label Propagation, initializing each node with its own label and then iteratively evaluating all nodes. Suppose we are currently evaluating the node u. For a given label l, let $N_u(l)$ bet the number of neighbors of u with label l. Let $T(l)$ be the total number of nodes in the graph with label l. The following quality function is then used to evaluate the label l:

$$f_u(l) = N_u(l) - \alpha \cdot (T(l) - N_u(l)) \tag{3.1}$$

where α is a parameter of the algorithm. Instead of simple taking the node that the maximum number of neighbors have, we take the label that maximizes Equation 3.1. Intuitively, the quality function chooses the most common label, but penalizes for all nodes with the same label that are not neighbors. This ensures that the detected clusters are well-connected, while the simple LP algorithm may output clusters that are internally disconnected. The parameter α adjusts the penalty for other nodes with the label. If α is large, only very dense clusters are detected. If α is small, clusters may be weakly connected, and APM equals LP for $\alpha = 0$.

Modularity

Modularity [CNM04] is a measure that compares the probability of an edge between two nodes i and j with the actual number of edges A_{ij} for a given clustering. The expected number of edges is given by a null model (the so-called configuration model [MR95]) in which each edge is cut in half to create two *stubs*, and then the stubs are randomly reconnected. Let e_c be the number of edges in cluster c. The expected number of edges is $\frac{K_c^2}{2m}$, where K_c is the sum of the degrees of the nodes in c. The modularity Q of a clustering C is given by

$$Q(C) = \frac{1}{2m} \sum_{c \in C} \left(e_c - \frac{{K_c}^2}{2m} \right) .$$

The value of the modularity lies in the range $[-1, 1]$. A higher value indicates that the clusters are better connected than expected from the null model. Networks with high modularity have dense connections inside the clusters, but sparse connections between the clusters. Modularity suffers from a *resolution limit* [FB07], which means the size of the detected clusters depends on the size of the whole network. On a large graph, modularity is not able to detect small clusters, because merging two small clusters may increase the modularity even if they are clearly distinct. Introducing a resolution parameter γ [RB06] is an approach to solve the resolution limit problem. The modularity is then given by

$$Q(C) = \frac{1}{2m} \sum_{c \in C} \left(e_c - \gamma \frac{{K_c}^2}{2m} \right) .$$

Higher resolutions lead to more and smaller clusters, while lower resolutions lead to fewer and larger clusters.

Constant Potts Model

An alternative quality function, similar to modularity, is the *Constant Potts Model* (CPM) [TVDN11]. The quality function of the CPM is given by

$$Q = \sum_{c_i} \left(e_c - \gamma \binom{n_{c_i}}{2} \right)$$

where n_{c_i} is the number of nodes in cluster c_i and γ is a resolution parameter. The parameter γ functions as a threshold for the detected clusters: A cluster should have a density of at least γ and the density between clusters should be lower than γ. A higher resolution lead to more clusters and a lower resolution lead to fewer clusters.

Louvain

Both modularity and CPM provide a quality function that can be optimized. However, finding the clustering with the highest quality is a NP-hard problem, so algorithms that approximate optimal solutions are necessary. The *Louvain* algorithm [BGLL08] was originally defined for modularity, but it can also be used to optimize other quality functions. The algorithm starts from a singleton clustering in which each node is in its own cluster. The algorithm has two phases that are repeated iteratively. Given a node u, the change in modularity is tested if it would be moved into the cluster of one of its neighbors. The node u is then moved into the cluster with the maximum modularity increase, if the increase is positive. This is done repeatedly for all nodes in the graph. The first phase stops when the modularity is locally optimal, i.e. no node can be moved to increase the modularity. The second phase creates an aggregated network, where each cluster is merged into a single node. Edges between clusters are preserved, i.e. an edge between two clusters in the aggregated network has a weight equal to the number of edges between the nodes of the two clusters. Internal edges are represented by self-loops. On this aggregated network, the modularity is again optimized locally. The two phases are repeated until the quality cannot be increased any further, i.e. no local moves are possible on the aggregated network. The resulting clusters are the nodes of the last aggregated graph, each representing a set of nodes.

Leiden

The *Leiden* algorithm [TWvE18] uses an approach similar to the Louvain algorithm. It is supposed to correct several shortcomings of the Louvain algorithm, e.g. that clusters may be internally disconnected. The Leiden algorithm is also based on local optimization and aggregation, but it is considerably more complex than the Louvain algorithm. It uses a fast local move procedure that only revisits nodes if their neighborhood changed in the last iteration. After the local optimization stops, an additional refinement step is introduced. In the refinement step, each cluster may be split into multiple sub-clusters. In the aggregated graph, each sub-cluster is then aggregated to one node. The initial partition for the aggregate network is based on the unrefined clustering, i.e. given a cluster c, all sub-clusters of c belong to c in the aggregated graph. This keeps the information of the local optimization, but gives the algorithm more room for identifying high-quality clusterings. The Leiden algorithm gives several guarantees for the clusters found, e.g. that clusters are γ-connected, where γ is the resolution parameter of either modularity or CPM. The Leiden algorithm is also faster than the Louvain algorithm in practical applications [TWvE18].

Infomap

The *Infomap* algorithm introduced by Rosvall et al. [RAB09] is based on the *map equation* [RB08]. In contrast to modularity and CPM, which are based on the concept of a null model, the map equation is based on a flow model. Suppose we have a random walker on the graph, and we want to describe its movement across the nodes with the minimal information (description length). Instead of assigning a unique code to each node, we partition the network into clusters. Each cluster has a unique code, but the code for the nodes can be reused across clusters, e.g. multiple clusters may have a node with code 10. If the random walker moves inside a cluster, the description length is shorter, because we only need to encode the nodes inside the cluster. If the random walker moves from one cluster to another, we have to use a special code that indicates the exit from the cluster, and another special code to indicate the entry into a cluster. The entry code is identical to the cluster code. The exit code is not the same across clusters, so a small cluster can use a shorter exit code. The goal of the Infomap algorithm is to compress the information, so that we can describe the movement of a random walker with the minimal description length. A weakly connected would often be exited, meaning we need to describe the exit and the entry into another cluster. Small clusters are preferable to large ones, as the number of nodes inside a cluster increase the length of the code for each node. Intuitively, a random walker is likely to stay inside a cluster, and rarely moves between clusters. This means that a clustering with the minimal description length gives us clusters that are strongly connected internally and weakly connected to other clusters.

The map equation for a clustering C can be written as [HSWZ18]

$$L(C) = \text{plogp}\left(\sum_{c\in C}\frac{K_c^{out}}{K_V}\right) - 2\sum_{c\in C}\text{plogp}\left(\frac{K_c^{out}}{K_V}\right) + \sum_{c\in C}\text{plogp}\left(\frac{K_c^{out}+K_c}{K_V}\right) + \sum_{v\in V}\text{plogp}\left(\frac{k_v}{K_V}\right),$$

where $\text{plogp}(x) := x\log x$.

3.2 Overlapping Community Detection

The overlapping community detection problem allows that one node is part of multiple communities. This problem is considerably more difficult than the non-overlapping problem and requires completely new approaches.

GCE

The *Greedy Clique Expansion* algorithm (GCE) [LRMH10] identifies distinct cliques as seeds and expands these seeds by greedily optimizing a local fitness function. The fitness function compares the internal degree of community to the external degree [LFK09]. Given a community S with total degree K_S and internal degree K_S^{in}, the community fitness is given by

$$F_S = \frac{K_S^{in}}{{K_S}^{\alpha}}.$$

The parameter α is a positive real number that can be tuned. In the experiments of Lee et al. [LRMH10], values for α between 0.9 and 1.5 provided the best results.

The algorithm starts from a clique S (a fully connected sub-graph of G). For all neighbors v of S, the fitness of v is calculated, i.e. how much the addition of v to S would raise the fitness function. Let v_{max} be the node with the largest fitness. If $v_{max} > 0$, add it to S and

reevaluate all neighbors, including any new neighbors of v_{max}. If $vmax \leq 0$, the algorithm terminates and returns S.

GCE uses as seeds all maximal cliques with at least k nodes, e.g. $k = 4$. After expanding a clique to a community c, c is compared to all other previously found communities c'. If c is too similar to a c' it is discarded, otherwise it is accepted. The similarity of two communities is defined as

$$\delta_E(c, c') = 1 - \frac{|c \cap c'|}{\min(|c|, |c'|)}$$

where lower values mean that the clusters are more similar. c is discarded if $\delta_E(c, c') < \epsilon$ for a given parameter ϵ.

OSLOM

The *OSLOM* algorithm (Order Statistics Local Optimization Method) [LRRF11] is based on the concept of *statistical significance* [LRR10]. The statistical significance evaluates the connection of a node i to a given community c. The statistical significance uses a null-model in which the edges inside c are locked, but all other edges are cut into to stubs and are then randomly reconnected. Each node retains its original degree. The null-model does not allow self-loops. The node i is statistically significant if it is unlikely that there is a node j in the null model with a stronger connection to c. In other words, i is stronger connected to c as we would expect from the null model. The probability that a node from the null model has a connection equal to or stronger than the connection of i is given by a value $r(i)$. OSLOM uses order statistics to compare $r(i)$ to the best expected node j from the null model, i.e. the node j with the lowest value $r(j)$. At the core of OSLOM is the single cluster analysis, which adds significant nodes to a community and removes all insignificant nodes. The single cluster analysis takes a community c and "cleans up" c, returning a significant community. OSLOM starts by creating a community from a single node, adding a number of neighbors at random and then using the cluster analysis to create a significant community. This is repeated several times, yielding a set of communities. OSLOM can also start from a given cover C. In this case, all communities of C are cleaned up using the cluster analysis. In both cases, OSLOM tries to find significant sub-communities. If two communities are too similar, they are merged together if the resulting community has a higher significance. OSLOM is also able to generate a hierarchical cover by detecting significant communities and then detecting significant sub-communities.

MOSES

The *MOSES* (Model-based Overlapping Seed ExpanSion) algorithm [MH10] is based on the concept of a *Stochastic Block Model* (SBM) [MH10]. The SBM is characterized by a set of parameters. In the simplest form, the SBM takes a cover of the graph with r communities and a symmetric $r \times r$ matrix of edge probabilities between the communities. The graph G is considered to be a realization of a statistical model, namely an instance of the SBM. One way to create a cover is finding the parameters that result in the maximum posterior probability, so the parameters that are most likely to generate G. MOSES uses an *Overlapping* Stochastic Block Model (OSBM) [LBA09] to include the possibility of overlapping communities. Their model makes assumptions about the graph, e.g. it does not explicitly model degree distribution. MOSES derives a fitness function from the model and then greedily expands communities based on this fitness function. Initially, edges are selected at random and a community is expanded around the two nodes of the edge. The expansion continues even if the fitness function decreases and stops when l consecutive expansions fail to raise the fitness function. The authors use $l = 2$ for their experiments. Periodically all the communities are scanned to see if the removal of an entire community

would increase the fitness. At the end each node is examined in a fine-tuning phase. The node is removed from all communities and then added again to communities if this raises the fitness.

Peacock

The *Peacock* algorithm [Gre09] tries to transform the overlapping problem into a non-overlapping one. The algorithm is based on GN. Gregory et al. [Gre07] introduces the concept of *split betweenness* to split vertices between communities. The split betweenness of a node v is the number of shortest path that would pass between the two parts of v if it were split. If a node is part of multiple communities, its split betweenness is expected to be high. The Peacock algorithm first tries to transform the network into a larger one that does not contain any overlapping communities. The transformation iteratively splits nodes as follows: Calculate the split betweenness of all nodes. Choose the node with the maximum split betweenness and split it into two, according to the best split. The best split is the one that maximizes the split betweenness. This is repeated until the maximum split betweenness is sufficiently small (a parameter of the algorithm). The split betweenness has to be recalculated after each iteration, but only for some nodes that are affected by the last split. At the end, for each split node, place an edge between the two resulting nodes. These edges are inserted so that the graph does not break into disconnected components, which would affect the communities that can be found on the transformed graph. On this new network, a non-overlapping community detection algorithm is executed. Each node is then assigned all communities of nodes that the original node was split into.

Ego-Nets

The concept of ego networks, or *ego-nets* for short, was first introduced by Freeman [Fre82]. The ego-net of a node u is the sub-graph induced by all neighbors of u. While the traditional definition also contains u, this adds "noise" to the ego-net [BKA+14] while providing no usable information. The study of ego-nets has established itself as a tool for social network analysis [Bur09, EB05, WF+94], providing useful information on a microscopic level.
Rees and Gallagher [RG10] were the first to analyze the ego-net structure to find global communities. They find communities in the ego-net by simply computing connected components. The global communities are obtained by merging significantly overlapping ego-net communities. The *DEMON* algorithm [CRGP14] advances this technique by using label propagation to find the ego-net communities.
Buzun et al. [BKA+14] use ego-net communities detection in combination with a global label propagation algorithm. The local communities influence the way the labels are propagated in the global graph.
Liakos et al. [LND16] iteratively merge nodes in the ego-net, yielding a hierarchical partition. This partition is then cut at the "right" level to get ego-net communities. They focus purely on finding communities for a given node, and do not create a global cover.
Soundarjan and Hopcroft [SH15] partition the ego-net and then create a new graph H where each node represents a detected ego-net community. They call such a ego-net community a *sub-community.* Two nodes in H are connected if their associated sub-communities in G are related in some way, e.g. their *Jaccard similarity* is larger than a given threshold. The Jaccard similarity of two communities c, c' is given by $\frac{|c \cap c'|}{|c \cup c'|}$. Then they detect communities in H using a clustering algorithm. Finally, the communities in H are converted into communities in G by using the union of all nodes in the sub-communities. The clustering algorithms and the metric to connect the nodes of H can be freely chosen.
Epasto et al. [ELPL17] partition the ego-nets using a clustering algorithm. Each node is then split into its *persona* nodes that represent the instantiations of the node in its communities. The personas of all nodes form the persona graph which only contains

non-overlapping communities. Then a clustering algorithm is applied to the persona graph, yielding a set of non-overlapping communities. Each node is then assigned to all communities of its personas, resulting in a cover of the original graph. The local and global clustering algorithm can be freely chosen.

4. The Ego-Splitting Framework

[ELPL17] introduced the *ego-splitting framework*. In the first step, the ego-net of each node is partitioned using a clustering algorithm. If we analyze the ego-net of u, we call u the *ego-node*. For each detected cluster, the node is split into a persona node, a copy of the node that is associated with the nodes of the corresponding cluster. The graph is then transformed into the persona graph $G' = (V', E')$ that uses the personas as nodes and maps each edge in E to one pair of personas. In the second step, a clustering algorithm is used to obtain non-overlapping communities on the persona graph. At the end, each node is assigned all communities that its personas are part of, resulting in overlapping communities on the original graph.

More formally, the algorithm requires two clustering algorithms $\mathcal{A}^l$ (the local clustering algorithm) and $\mathcal{A}^g$ (the global clustering algorithm). Let G_u be the ego-net of node u. The following five steps create the cover of overlapping communities:

- *Step 1*: For each node u, use the local clustering algorithm to partition the ego-net. $\mathcal{A}^l(G_u) = \{N_u^1, N_u^2, \ldots, N_u^{t_u}\}$
- *Step 2*: Create a set V' of personas. Each node corresponds to t_u personas denoted as u_i, $i = 1, \ldots, t_u$.
- *Step 3*: Add edges between the personas. For each edge $(u, v) \in V$, find u_i, v_j such that $v \in N_u^i$ and $u \in N_v^j$. Add (u_i, v_j) to E'.
- *Step 4*: Apply the global partition algorithm to obtain clusters on G'. $\mathcal{A}^g(G') = C'$.

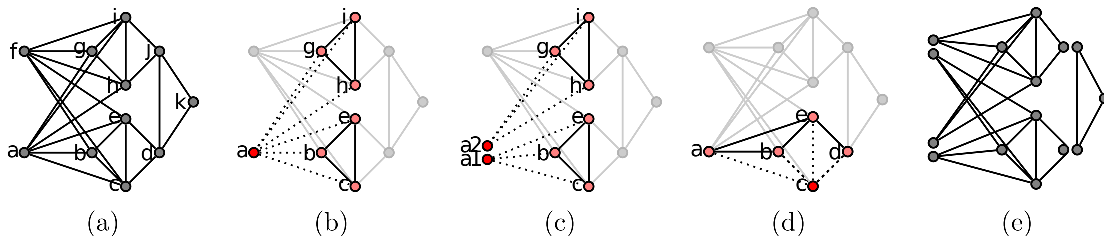
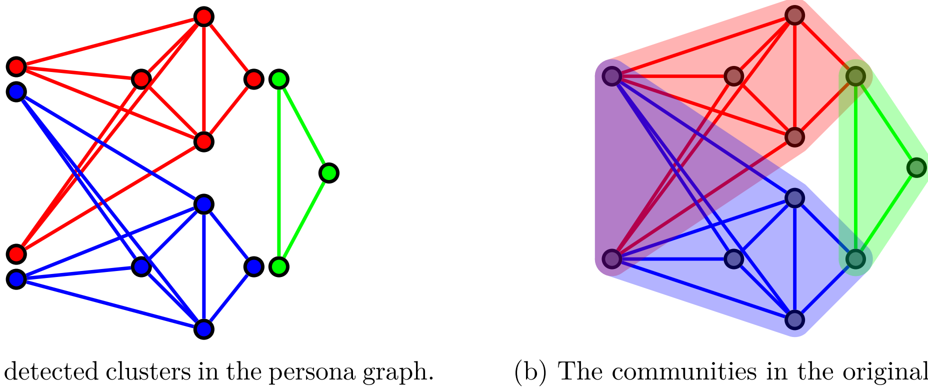


Figure 4.1: Partitioning the ego-nets, splitting the nodes into personas, and building the persona graph. Illustration by Epasto et al. [ELPL17]. (a) shows the original graph G. The ego-net of node a consist of two clusters (b), so a is split into two personas a_1, a_2 (c). The ego-net of node c has only one cluster, so c has only one persona c_1 (d). After splitting all nodes, the persona graph is created (e).



(a) The detected clusters in the persona graph. (b) The communities in the original graph.

Figure 4.2: Clustering of the persona graph and corresponding communities in the original graph. Illustration by Epasto et al. [ELPL17]

- *Step 5*: Create the communities in G. For each cluster $c' \in C'$, create the associated community $c \subseteq V$ formed by the corresponding nodes of V: $c(c') = \{u \in V | \exists i \text{ s.t. } u_i \in c'\}$. The result cover of the framework is $C = \{c(c') | c' \in C'\}$.

Figure 4.1 illustrates steps 1 to 3 on a small sample graph, using connected components as the local clustering algorithm. Figure 4.1b depicts Step 1 for node a: We get the ego-net of a and partition it into two clusters. In Figure 4.1c(c), one persona is created for each detected cluster. For example, a_1 is associated with the nodes b, c, e. In Figure 4.1d, we do the same for node c. As there is only one cluster detected, we create only one persona c_1 that is associated to all neighbors a, b, d, e. After this is done for all nodes, we build the persona graph by adding the edges of G (Step 3). For example, the edge (a, c) in the original graph is inserted into the persona graph as an edge between the persona of a associated with c (a_1) and the persona of c associated with a (c_1). Step 4 is shown in Figure 4.2a where we apply the global clustering algorithm on the persona graph. Finally, we perform Step 5, depicted in Figure 4.2b, where we map the persona communities to the nodes of the original graph.

Intuitively, we find all communities a node u is part of. Then we split u into multiple personas, each corresponding to one of its communities. Each persona obtains the edges between u and nodes of its associated community. Each persona is then only part of one community, so the global clustering algorithm is able to detect all communities on the persona graph. Transforming the graph into the persona graph increases the number of nodes, but it keeps the number of edges constant. Since there is a one-to-one mapping of the edge in both graphs, the clustering of the persona graph also assigns a community (or none) to each edge. In this sense, the framework also produces an edge partition (also called link partition).

A naive way to calculate all ego-nets could be computationally expensive, in the order of $O(mn)$. An optimal algorithm can create all ego-nets in time $O(m^{3/2})$. In practice, the upper bound depends directly on the number of triangles in the graph.

Noticeable is that the framework does not rely on any specific clustering algorithms. It can be adapted to different requirements or specific graphs by choosing appropriate clustering algorithms. The framework is able to handle weighted and/or directed graphs, provided that $\mathcal{A}^l$ and $\mathcal{A}^g$ also support this. The authors use a label propagation algorithm based on the Absolute Potts Model [RN10] as the local and global clustering algorithm in their experiments, because the algorithm is fast and can be run in parallel.

The ego-splitting framework relies on the fact that each node is split correctly into its personas. This means that the local clustering is a critical part of the algorithm. If a node

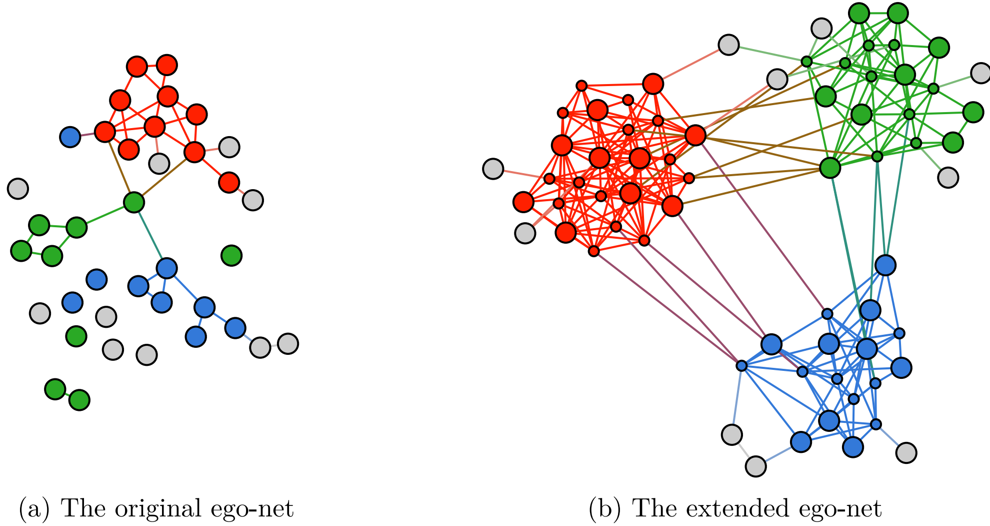
is split into too few personas, it cannot be assigned to all of its communities, no matter how high the quality of the global clustering algorithm is. If the local clustering algorithm is not able to partition the ego-net into the communities of the ego-node, then the overlapping communities are not properly entangled. If the persona graph still contains overlapping communities, the global clustering algorithm will not be able to detect the communities correctly. In this thesis, we will evaluate some problems of the framework in detail and present extension of the framework to improve the quality of the detected cover.

5. Ego-Net Extension

The clustering of the ego-net is critical for the quality of the end result of the ego-splitting framework. The idea of the framework is to disentangle the overlapping communities by splitting each node in a set of personas such that each persona is associated with one community. The communities can only be disentangled well if the local clustering algorithm provides a high quality clustering, i.e. each community is detected as exactly one cluster. A low quality clustering can cause multiple problems in the persona graph. If the clustering algorithm detects to many clusters, too many personas are created. In the persona graph, a community may then be decomposed into multiple components, or be only sparsely connected. On the other hand, if the clustering algorithm detects too few clusters, the communities are not properly disentangled. In this case, the persona graph still contains overlapping communities. If a node from community a is assigned to the cluster of community b, then community b will contain more inter-community edges in the persona graph, while community a will contain less intra-community edges. In all cases, the persona graph does not properly represent the disentangled communities, so the global clustering algorithm will most likely not detect all communities correctly.

Let the node u be part of the ground-truth communities $c_i, i = 1, \ldots, t$. When we look at the ego-net G_u of u, some nodes inside the ego-net are part of one of the c_i, while other nodes are not. A node that is not part of any of the c_i is called an *external* node in the context of the ego-net. Each ground-truth community c_i induces a sub-community c_i^u in the ego-net of node u, $c_i^u = \{v \in G_u | v \in c_i\}$. In the following analysis, we assume that the c_i^u are disjoint. In reality, the c_i^u may overlap, but only to a small degree. The optimal output of the local clustering algorithm would detect each c_i^u as one cluster and put each external node inside a singleton cluster. Each created persona of the ego-node would then correspond to a ground-truth community or an external node. We call c_i^u a *ego-community* of node u.

In preliminary experiments, we visualized some ego-nets and found that frequently, there are nodes that are disconnected or sparsely connected to their ego-community. In general, there is no guarantee that the nodes of a ego-community c_i^u form a detectable cluster in G_u. For example, c_i^u may be internally disconnected, i.e. the nodes form sub-sets that are disconnected. In this case, it is unlikely that c_i^u will be detected as a single cluster, as we expect any reasonable clustering algorithm to put two disconnected components into two separate clusters. Figure 5.1a shows an example of an ego-net in which the green community is split into multiple components. The problem lies in the structure of the ego-net itself and not in the quality of the clustering algorithm. We use the term *structural*



(a) The original ego-net (b) The extended ego-net

Figure 5.1: Comparison of the ego-net of a node u before and after extension. The graphs were drawn using a force-directed layout. Large nodes represent nodes of the original ego-net, small nodes represent nodes that were added by the extension. The colors indicate ground-truth communities of u: Green, red, and blue nodes are part of the same community. Gray nodes are in none of the ground-truth communities (external nodes).

quality to assess the structure of the ego-net. The structural quality indicates how easy it is to detect the ego-communities as clusters in the ego-net. If the structural quality is low, even the best clustering algorithm cannot detect the communities. Vice versa, increasing the structural quality should increase the quality of the detected clusters for most reasonable clustering algorithms. We use structural quality as a generic term. To measure the structural quality, we have to use specific metrics, e.g. the conductance of the communities or the ratio of intra-community edges to inter-community edges. If we increase the structural quality considerably, most reasonable clustering algorithms should produce a higher quality clustering. One possible approach to increase the structural quality is the inclusion of meta-data. However, this is obviously only possible if such meta-data exist and if the meta-data actually induces the communities we want to find, so this approach is only useful in specific cases.

In preliminary experiments, we found that ego-nets are often very sparse, see Figure 5.1a for an example. The colors indicate the communities c_i^u, and gray nodes are external nodes. Note how the green and blue community are internally disconnected and there are many isolated nodes. We propose an extension of the ego-net to increase the structural quality. Extending the ego-net means that we add additional nodes to the ego-net, i.e. nodes of G that are not direct neighbors of the ego-node. Our goal is to add only nodes that are part of one of the ground-truth communities c_i that we want to detect in the ego-net. For example, if we add a node $v \in c_i$, we expect that the structural strength of c_i increases. Figure 5.1b gives an example of the extended ego-net. Both graphs were drawn by a force-directed layout. While the communities in the original ego-net are internally disconnected, they are well-connected in the extended ego-net. At the same time, the communities are better separated. The force-directed layout splits the nodes into three clearly separated sets of nodes that correspond to the ground-truth communities. We expect that most clustering algorithms are able to detect a higher quality clustering in the extended ego-net.

To decide which nodes we add to the ego-net, we first have to get a set of candidates $J \in G \backslash (G_u \cup \{u\})$. We use as candidates all neighbors of the nodes in the ego-net, i.e. all

nodes that are connected to the ego-node by a path of length 2 (neighbors of neighbors of the ego-node). In most graphs, a node has many more neighbors of neighbors than direct neighbors, so the number of candidates is far higher than the size of the ego-net. There are two reasons to keep the number of added candidates low: First, most of the candidates are not part of the communities we want to find. Adding all of them would most likely decrease the structural quality. We want to add only candidates that improve the structural quality of the ego-net, so we need to evaluate the candidates and add only the best ones. Second, adding nodes to the ego-net also increases the running time of the local clustering algorithm. Adding all candidates to ego-net may increase the complexity of the entire ego-splitting framework considerably. We limit the number of nodes that are added to the ego-net. Since the ego-net sizes vary greatly, we choose this limit dependent on the size of the ego-net. Let n_e be the number of nodes in the ego-net. The limit for the number of added nodes is $o = \alpha \cdot {n_e}^{\beta}$. In practice, β should be smaller or equal to 1 in most cases, as this means that the ego-net is at most extended by a constant factor.

We present two approaches to evaluate the candidates, which we will describe in the following sections. The first approach rates candidates with a simple scoring function based on the number of neighbors in the ego-net. The second approach is more sophisticated and is based on the statistical significance of the candidate.

5.1 Number of Edges

A simple way to rate the candidates is to count the number of edges into the ego-net. Let $j \in J$ be a candidate that we want to evaluate. If j is in one of the ego-communities, we expect a comparatively large number of edges between j and the nodes of the ego-community. Vice versa, an external node has a lower chance to be well connected to the ego-net. This implies that a node with many edges into the ego-net has a higher chance to be in one of the ego-communities than a node with fewer edges.

For each node $v \in G_u$, we look at all of its neighbors w. If $w \in G \backslash (G_u \cup \{u\})$, we mark w as a candidate and increment a counter k_w^{in}, starting from 0. After this process is done for all nodes in the ego-net, k_w^{in} gives the number of edges to the ego-net for each candidate w. Using this value, we define a scoring function $q_1(j) = k_j^{in}$ that gives us a score for each candidate. We calculate the score for each candidate and then extend the ego-net by adding the o best candidates to it. However, q_1 does not take into account that the candidates may have different degrees. For example, let j and j' be two candidates with $k_j^{in} = k_{j'}^{in}$ and $k_j > k_{j'}$. Intuitively, j is a better candidate than j because j has a higher fraction of its edges going into the ego-net. High degree nodes have a higher chance to have "random" edges into the ego-net than low degree nodes. To facilitate the comparison of nodes with different degrees, we can normalize k_j^{in} by dividing it by the node degree of j. This gives us the candidate score function $q_2(j) = \frac{k_j^{in}}{k_j}$. However, q_2 is not optimal in all cases. Suppose we have two candidates j and j' with $k_j^{in} = 20, k_j = 40$ and $k_{j'}^{in} = 5, k_{j'} = 10$. The fitness of these candidates is $q_2(j) = q_2(j') = 0.5$. We would prefer no candidate over the other, but there are two reasons why we want to prefer j over j': First, j is stronger from a stochastic point of view. In large networks, the ego-net is much smaller than the rest of the graph. If an edge (j, v) is drawn at random from all $v \in G$, the probability that v is a node of the ego-net is $p = \frac{|G_u|}{|G|} \ll 0.5$. Let us assume that j, j' are externals nodes with a given chance to have an edge to any node of the ego-net. It is much more unlikely that j has 20 of its 40 randomly connected to the ego-net than that j' has 5 of its 10 randomly connected to the ego-net. Second, as j has more connections into the ego-net, adding j would increase the number of edges inside the ego-net much more than adding j'. If j is an internal node, most of these edges are intra-community edges. j' would most likely result

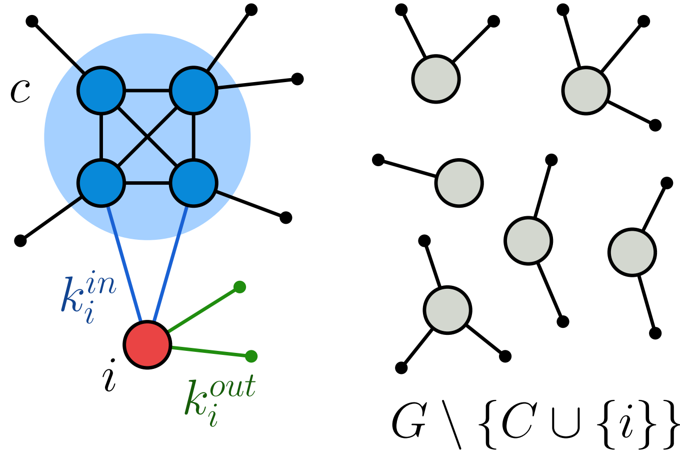


Figure 5.2: Evaluating the statistical significance of a node i to a community c. In this example, $K_c^{in} = 6, K_c^{out} = 7, M = 13, M^{in} = 6, k_i^{in} = 2, k_i^{out} = 2$.

in fewer intra-community edges, so j increases the structural quality further. We propose the scoring function

$$q_e(j) = \frac{{k_j^{in}}^2}{k_j}\,.$$

This scoring function prefers candidates with many edges, but still has a penalty for high degree nodes. The extension process is straightforward: We calculate the score $q_e(j)$ for each candidate $j \in J$. Then we sort the candidates by their score and add the o best candidates to the ego-net. To improve the running time, we ignore all candidates with $k_j^{in} < 3$. Because such a candidate only adds very few edges to the ego-net, we can assume that adding it would not considerably improve the ego-net structure.

5.2 Statistical Significance

A more sophisticated approach to rate the candidates is based on statistical significance [LRR10]. The statistical significance evaluates one node i and its edges to a given community c in G ($i \notin c$). The evaluation is based on a null-model, i.e. a class of graphs without community structure. In the null model, edges inside c are locked, but all other edges are randomly reconnected. Each node thus retains its original degree. The null-model does not allow self-loops. The statistical significance of i is based on the probability that the neighbors of i were randomly drawn by the null model. Intuitively, if we reconnect the edges of i according to the null model, then we can calculate the expected number of neighbors in c. If i has much more neighbors in c than expected, we can assume that i is not randomly connected to c. Instead, it has a strong connection to c (or a subset of c) which indicates that it is part of one of the ego-communities.

Let c be a community in G and i be the node which significance we want to assess. Figure 5.2 depicts the situation. The number of neighbors of i inside c is denoted as k_i^{in}, the neighbors in the rest of G as $k_i^{out} = k_i - k_i^{in}$. The degree of c is K_c, which can be separated into the internal degree K_c^{in} and the outgoing stubs K_c^{out}. The total degree of $G \setminus \{c \cup i\}$ is M and its internal degree is M^{in}. M^{in} is the number of stubs that are not connected to either c or i, $M^{in} = M - (K_c^{out} - k_c^{out} + k_c^{in})$. Given such a community c and a node i, the probability that i has exactly k_{in} neighbors in c according to the null-model is given by

$$p(k_i^{in}) = A \frac{2^{-k_i^{in}}}{k_i^{out}! k_i^{in}! (K_c^{out} - k_i^{in})! (M^{in}/2)!}\,. \tag{5.1}$$

The normalization factor A ensures that

$$\sum_{x=0}^{k_i} p(x) = 1\,.$$

The normalization factor is important to get the correct result, but it cannot be easily approximated. To calculate it, we would have to use equation 5.1 and evaluate it for every possible k_i^{in}. We have to calculate this many times, so using this method would result in a very high complexity. Lancichinetti et al. [LRRF11] propose to approximate the distribution with another distribution that is easier to estimate. Suppose we would allow self-loops in the null model. Then we would actually have the same null model as the one on which the definition of modularity is based [New06]. In such a null model, the equivalent to equation 5.1 can be calculated by a hypergeometric function. We can then directly calculate the value without a normalization factor by using the equation

$$p(k_i^{in}) = \frac{\binom{K_c^{out}}{k_i^{in}}\binom{M}{k_i^{out}}}{\binom{M+K_c^{out}}{k_i}}\,.$$

The equation gives the number of ways the stubs can be connected for the given k_i^{in}, divided by the total number of ways to place all k_i. A hypergeometric distribution describes the probability of drawing a given number of successes without replacement. For each of the k_i edges, we draw at random an open stub. The total number of possible stubs is $M + K_c^{out}$ and the number of "successful" stubs is K_c^{out}. We calculate the probability of exactly k_i^{in} successful draws.

In most cases, the change in the null model does not have a large impact on the result of the equations, so the hypergeometric distribution is a good approximation of our actual distribution. However, if the probability of generating self-loops in the modified null model is high, the null model and the modified null model differ considerably. In this case, the null models may produce widely different graphs, so our approximation is not reliably good. In the modified null model, the probability that a random stub of node i connects to another stub of the same node is given by $\frac{{k_i}^2}{2M}$. If ${k_i}^2 > 2M$, the expected number of loops is larger than 1. In this case we calculate Equation 5.1 directly. For the network graphs we evaluated, the equation ${k_i}^2 < 2M$ holds true for all nodes.

Equation 5.1 gives the probability that i has exactly k_i^{in} neighbors in c according to the null model. This value alone however is not comparable for different nodes, as a higher degree always decreases the value for a single k_i^{in}. Also, the probability is not monotonically decreasing, e.g. the chance that a node has 0 neighbors in the ego-net may be smaller than the chance that is has 1 neighbor. Therefore, we have to calculate the probability that i has at least k_i^{in} neighbors according to the null model. This probability is given by

$$r(k_i^{in}) = \sum_{x=k_i^{in}}^{k_i} p(x)\,.$$

We will refer to this probability as the *r-score*. Let r_i be the r-score of i. The r-score provides a tool to evaluate the topological relation of i and c. The r-score indicates how likely it is that i has k_i^{in} or more neighbors in c according to the null model. If i has many more neighbors in c than expected in the null model, the r-score is low. In other words: If a node has a low r-score, its connection to c is unexpectedly strong.

The r-score does not consider the number of external nodes, only how likely a single node from the null model is to have at least k_i^{in} neighbors in c. For example, if $r_i = 0.01$, we

Algorithm 5.1: EXTENDSIGNIFICANCE

Input: Graph G, Ego-Net G_u of u, Clustering of D of G_u, limit of added candidates o
Data: List of candidates J, Edges between candidates and clusters e
Output: Set of significant candidates J'

```
1 J, e ← searchCandidates(G, G_u, D)
2 J' ← {}
3 forall j ∈ J do
4     if |J''| ≥ o then
5         return
6     s ← checkSignificance(j, D, e)
7     if s = true then
8         J' ← J' ∪ {j}
```

can interpret this as i having a chance of 1 to 100 to be a created by the null model. If we only have 10 external nodes, we might consider this good enough and conclude that the node is not compatible with the null model, because it has significantly more neighbors in c than expected. However, if there are 1000 external nodes, we expect that there are around $1000 \cdot 0.01 = 10$ nodes created by the null model with such an r-score. In this case, the r-score of i is not better than the r-score of an external node, so we conclude that i is compatible with null model.

Let $n_e = n - |c|$ be the number of external nodes and r_i be the r-score of the candidate i. r_i gives the probability that i has at least k_i^{in} neighbors in c according to the null model. We calculate the probability that from the n_e nodes there is at least one that has lower (better) r-score than r_i. If we look at one external node, the probability that its r-score is lower than r_i is, according to the null model, r_i. Examining all external nodes thus equals a binomial distribution with a success probability of r_i and a sample size of n_e. The probability that at least q of the n_e nodes have a r-score smaller than r_i is given by

$$\Omega_q(r_i, n_e) = \sum_{j=q}^{n_e} \binom{n_e}{j} {r_i}^j (1 - r_i)^{n_e - j} .$$

The special case for $q = 1$

$$\Omega_1(i) = 1 - (1 - r_i)^{n_e}$$

gives us the probability that among the n_e external nodes there is at least one with a better (lower) r-score than r_i. We define the *s-score* of a node i to a given community c as

$$s(i, c) = \Omega_1(r_i, n_e) .$$

The s-score expresses how likely it is that the neighbors of i haven been drawn randomly according to the null model. If i is part of a ego-community, it should have more neighbors in the ego-net than expected according to the null model, so its s-score is low. Vice versa, if i is an external node, we expect that k_i^{in} is in the range of expected values, so its s-score is high. We call a candidate *statistically significant* (or just *significant*) if its s-score is so low that we consider the candidate not compatible with the null model. More specifically, we have to set a tolerance P, $0 < P \ll 1$, for example $P = 0.1$. A node i is significant if $r_i < P$.

We use statistical significance to rate the candidates for the ego-net extension. We only add candidates that are significant, as these are likely to be part of an ego-community.

Algorithm 5.2: SEARCHCANDIDATES

Input: Graph $G = (V, E)$, Ego-Net $G_u = (V_u, E_u)$ of u, Clustering D
Output: Set of candidates J, Number of Edges $\mathsf{e}(w,\ d)$ between all $w \in \mathsf{J}$ and $d \in \mathsf{D}$

```
   // Initialization
 1 J ← {}
 2 forall w ∈ V do
 3 |  forall d ∈ D do
 4 |  |  e(w, d) ← 0

 5 forall v ∈ V_u do
 6 |  d ← D.clusterOf(v)
 7 |  forall (v, w) ∈ E do
 8 |  |  if w ∉ V_u then
 9 |  |  |  J.insert(w)
10 |  |  |  e(w, d) ← e(w, d) + 1
```

The significance is calculated in the global graph G. As described above, the definition of statistical significance is based on the connection of a node to a community. When we first build the ego-net, we do not have any information about the communities, so the only known community we could use is the whole ego-net. A simple approach would be to just calculate the significance of the candidate to the whole ego-net. However, if there are multiple communities in the ego-net, we expect that even a good candidate is only well-connected to a subset of the ego-net (usually just one community). If we just take the significance of the candidate to the whole ego-net, the s-score would decrease if the number of communities, and therefore the size of the ego-net, increases. This approach is only useful if the number of communities per node is very low (≤ 2). To deal with highly overlapping communities, we have to calculate the significance to each ego-community separately. To accomplish this, we need an estimation of the ego-communities, which is the same problem as finding a good clustering. Obviously, we cannot expect to get a perfect clustering, because we would not need to extend the ego-net in this case. Instead, we calculate the significance from an approximation of the real communities. To do this, we first cluster the ego-net, using the local clustering algorithm. We can either simply use the original ego-net, or we first extend the ego-net with another approach, e.g. using the number of edges as described above. As said before, the detected clusters are not an optimal fit to the communities we want to find. To keep it simple, we assume that each detected cluster mostly consists of nodes from one community, as splitting detected clusters is much harder than joining them. In our experiments, this assumption holds true for some clustering algorithms, e.g. modularity based algorithm tend to keep communities separate, but also split a single ground-truth community into multiple parts.

Algorithm 5.1 gives an overview of the algorithm to extend the ego-net. First, we apply the local clustering algorithm to receive a clustering D. Next, we search for extension candidates. The process is given in pseudo code in Algorithm 5.2. For each node $v \in G_u$, we look at all of its neighbors w. If $w \in G \setminus (G_u \cup \{u\})$, we mark w as a candidate. Let $d \in D$ be the detected cluster that v is part of. We increment a counter $e(w, d)$ to remember the edge between w and d. Then we evaluate all candidates, adding significant ones to the ego-net.

Algorithm 5.3 shows the process of evaluating the significance of a candidate. The evaluation of a candidate j consists of two phases. In the first phase, we calculate for j its s-score

Algorithm 5.3: CHECKSIGNIFICANCE

```
    Input: Candidate j, Clustering D, Number of Edges e(j, d) between j and all
           d ∈ D
    Data: Map S, List of Clusters D_s
    Output: true if j is significant, else false
    // Significance to clusters
 1  D_s ← CLUSTERSSORTEDBYEDGES(D, e, j)
 2  forall d ∈ D_s do
 3  |   s ← CALCSIGNIFICANCE(j, d, e(j, d))
 4  |   S(d) ← s
 5  |   if s < P then
 6  |   |   return true
    // Significance to merged clusters
 7  D_s ← CLUSTERSSORTEDBYSIGNIFICANCE(D, S)
 8  d_m ← {}
 9  e_m ← 0
10  forall d ∈ D_s do
11  |   d_m ← d_m ∪ d
12  |   e_m ← e_m + e(j, d)
13  |   if CALCSIGNIFICANCE(j, d_m, e_m) < P then
14  |   |   return true
15  return false
```

$s(j,d)$ to each detected cluster d. As described above, j is significant if $s(j,d) < P$ for a constant tolerance P. If we find a cluster that j is significant to, we add j to the ego-net and terminate the evaluation of j. To increase the efficiency of the algorithm, we first sort the clusters by the number of edges (connections) to the candidate. We expect that j is more likely to be significant to a cluster if it has more connections. Then we calculate the significance for each cluster, starting with the cluster with the highest number of connections.
If we examined all clusters without finding j to be significant, the second phase of the evaluation of j begins. Ideally, we would want to calculate the significance of j to all ego-communities c_i^u. As we do not have any reliable information, we consider j significant if it is significant to any sub-group of the ego-net For each cluster d, we store its fitness $s(j,d)$ after we have calculated it in the first phase. We sort the clusters by their fitness. Let $d_i, i = 1, \ldots, t$ be the detected clusters, sorted descending by their fitness, i.e. d_1 is the cluster with the best fitness. We create a new cluster by combining d_1 and d_2. Let $D_2 = d_1 \cup d_2$ be the merged cluster. Now we calculate the significance $s(j, D_2)$ to the new cluster. If $s(j, D_2) < P$, we add j to the ego-net and terminate the evaluation of j. Otherwise, we merge the next best detected cluster into D_2, resulting in $D_3 = D_2 \cup d_3$ and check the significance to D_3. We iteratively merge clusters this way, yielding $D_i = \bigcup_{j=1}^{i} d_i, i = 1, \ldots, t$. If there is no D_i with $s(j, D_i) < P$, we consider j not significant and discard it as a candidate.

The evaluation of a candidate as described above is repeated for each candidate, resulting in a set of significant candidates that are added to the ego-net. As described earlier, there is an upper bound $o = \alpha \cdot s^{\beta}$ for the number of nodes that we add. If this bound is reached after evaluating a candidate, we terminate the extension without evaluating the remaining candidates.

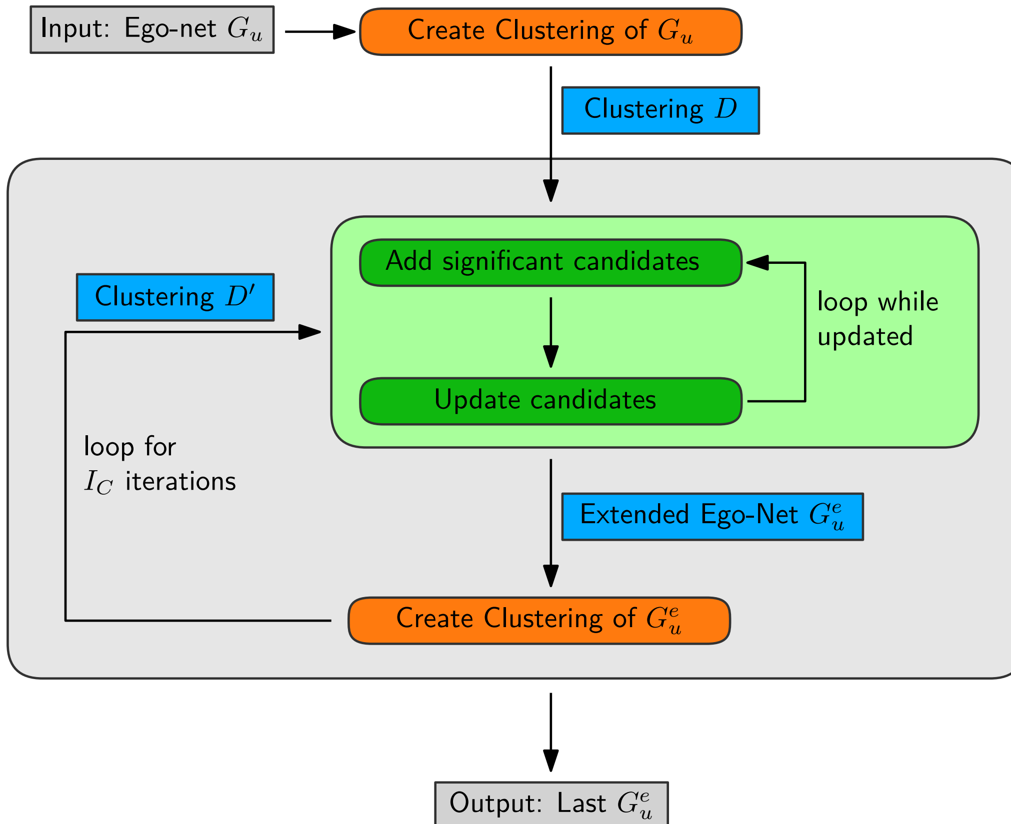


Figure 5.3: Complete extension algorithm based on statistical significance. The clusterings are created by applying the local cluster algorithm on the original/extended ego-net. Before the clustering of the extended ego-net is used, the ego-net is reverted back to the original, i.e. the process in the green box always starts from the original ego-net.

To get the best candidates, we would have to calculate the significance for every candidate. However, the calculation of the significance is relatively expensive, so we optimize the complexity of the candidate selection. First, we discard all candidates with less than three neighbors in the ego-net. It is unlikely that such a node would be significant, and even if it were, adding a node with only one or two edges would not improve the structure of the ego-net. This still leaves a great number of candidates which we have to evaluate. We have already defined an upper limit o of the number of nodes that we extend. We define the upper limit of evaluated candidates as $\bar{o} = \gamma \cdot o, \gamma \geq 1$. After searching for all candidates, we sort them by the number of neighbors they have in the ego-net, and then we take the $\bar{o}$ best. If we would evaluate all candidates, it would be difficult to make any performance guarantees. By using the upper limit, we evaluate at most $\gamma \cdot o \in O(n_e)$ candidates.

After the evaluation of all candidates, we have a set of significant candidates. The number of significant candidates may be still below the limit o. In this case, we try to add additional candidates to the ego-net. We also iteratively search for candidates, which we describe further below. Figure 5.3 gives an overview of the entire extension algorithm. Because we already added some candidates, the structure of the ego-net has changed. Suppose we have a ground-truth community c_i and its corresponding ego-community c_i^u. Now we add a significant candidate $j \in c_i$, so $c_i^u = c_i^u \cup \{j\}$. Suppose that there is a candidate $j' \in c_i$ that was not considered significant and j and j' are connected by an edge $(j, j') \in E$. If we recalculate the significance of j', the s-score is likely to be better than before, as j' has an additional neighbor in the ego-net. With this in mind, we can recalculate the

significance of each remaining candidate and add all candidates that are now significant. As the significance is calculated based on a cluster $d \in D$, we have to assign each extended candidate j to one of the clusters. If we find a single cluster d that j is significant to in the first phase of the candidate evaluation, we assign j to d ($d = d \cup \{j\}$). If j is not significant to a single cluster, we performed the second phase and examined the significance of j to a merged cluster $d_1 \cup d_2 \cup \cdots \cup d_x$. Because we sorted the clusters by their significance, d_1 is the cluster with the highest significance to j. If j is found to be significant in the second phase, we add j to d_1. Let d^j be the cluster that we have assigned j to. First, we update the information about the cluster d^j. The formula of the significance depends on three features of the inspected group: The number of outgoing stubs ($K_{d^j}^{out}$), the open stubs in the rest of the graph (M) and the number of nodes in the rest of the graph (n_e). We can update all values in constant time: $K_{d^j}^{out} = K_{d^j}^{out} - k_j^{in} + k_j^{out}, M = M - k_j, n_e = n_e - 1$. Then we update the remaining candidates as follows. For each edge $(j, j') \in E$, we check if $j' \in J$. If this is the case, we increment the number of edges $e(j', d^j)$ between the other candidate and the cluster of j by one. We repeat this update for all candidates that were added to the ego-net. Then we start the evaluation of all candidates, excluding these that were already added. This process is repeated iteratively until no new candidates are added. With the procedure described above, we reevaluate all candidates even if they did not receive additional edges into the ego-net. To lower the complexity of the extension, we only evaluate candidates that were updated at the end of the last iteration. We also set an upper bound I_{max} for the number of iterations.

As mentioned above, the evaluation of the significance is based on a clustering D. We calculate D by applying the local clustering algorithm on the original ego-net G_u. One problem is that the original ego-net may have a low structural quality, which we hope to improve with the ego-net extension. The successful detection of significant candidates relies partially on a good quality clustering. We assume that the structural quality of the ego-net increases with the extension. We make use of this by applying the local clustering algorithm on the extended ego-net G_u^{ext}, resulting in a new clustering D'. Then we revert the extended ego-net back to the original one. Consequently, we remove all nodes $j \notin G_u$ from the clusters of D', resulting in a clustering of the original ego-net. We can now use the clusters of D' as the basis for the significance calculation and repeat the extension procedure. This process can be iteratively performed, using the last extended ego-net as the input for the local clustering algorithm. Ideally, each iteration increases the quality of the clustering, which in turns makes the detection of significant candidates more precise. In our experiments, the extended ego-net did not change considerably anymore after a low number of iterations, e.g. 3.

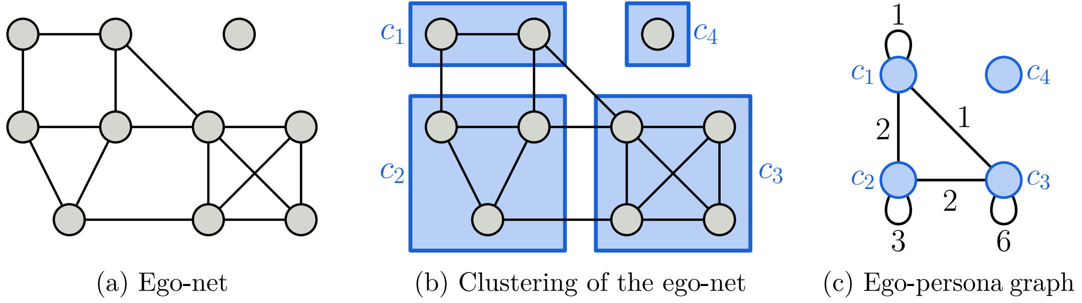
One disadvantage of Significance is its reliance on a clustering algorithm. First, this increases the running time of the extension process considerably. Second, if the local clustering algorithm provides only a low quality clustering, then we might not be able to find significant candidates.

6. Connecting Personas

After clustering the ego-net with the local clustering algorithm, a persona copy of the ego-node is created for each detected cluster. Then each edge of the original graph is inserted into the persona graph. Let $u_i, i = 1, \ldots, p$ be the personas of node u. In the persona graph, the u_i are not connected by any edges. Intuitively, this makes sense because each persona should correspond to exactly one community. By creating the personas, we entangle the communities that overlap at node u. However, this assumes that the assignment from personas to communities is exactly one-to-one. In this case, we have an optimal partitioning of the ego-net. In general, we cannot assume that the output of the local clustering algorithm is an optimal clustering. One error that the clustering algorithm can make is creating two (or more) cluster for a single ground-truth community c. If this happens, we create two personas u_1, u_2 that should belong to the same community c. In the persona graph, the edges between u and nodes of c are split between u_1 and u_2. The size of c in the persona graph increases by one compared to the optimal clustering. This makes c weaker, e.g. the density decreases. The global clustering algorithm has no information about u_1 and u_2, so it just handles them as two entirely independent nodes.

We propose to connect the personas of each node by inserting additional edges into the persona graph. This should improve the quality of the global clustering algorithm if the local clustering algorithm returns a non-optimal clustering of the ego-net. Connecting the personas retains the information that they are a single node in the original graph. If we don not connect them, the persona graph can break into disconnected components. If the local clustering algorithm would work perfectly, this would actually be a plus, because the global clustering algorithm would just have to take each connected component as one community. However, the local algorithm is in general imperfect, i.e. a detected cluster does not match a ground-truth community. When two (or more) communities are merged into a single cluster, the ego-node can be only assigned to one of them in the global graph, so its impossible for the global clustering algorithm to repair this error. When a community is split, the two personas can still end up in the same detected community, so the global clustering algorithm can (in principle) repair this kind of error. Connecting the personas helps the global clustering algorithm, detect this kind of error. For example, the internal density of the community increases because of the added edges.

The Peacock algorithm [Gre09] shares similarities with the ego-splitting framework. In the first phase of the algorithm, G is transformed into a graph H containing no overlapping communities. To create H, Peacock uses edge betweenness and split betweenness to repeatedly split nodes. In the second phase of Peacock, a clustering algorithm is applied to



(a) Ego-net (b) Clustering of the ego-net (c) Ego-persona graph

Figure 6.1: Create the ego-persona graph from the ego-net clustering.

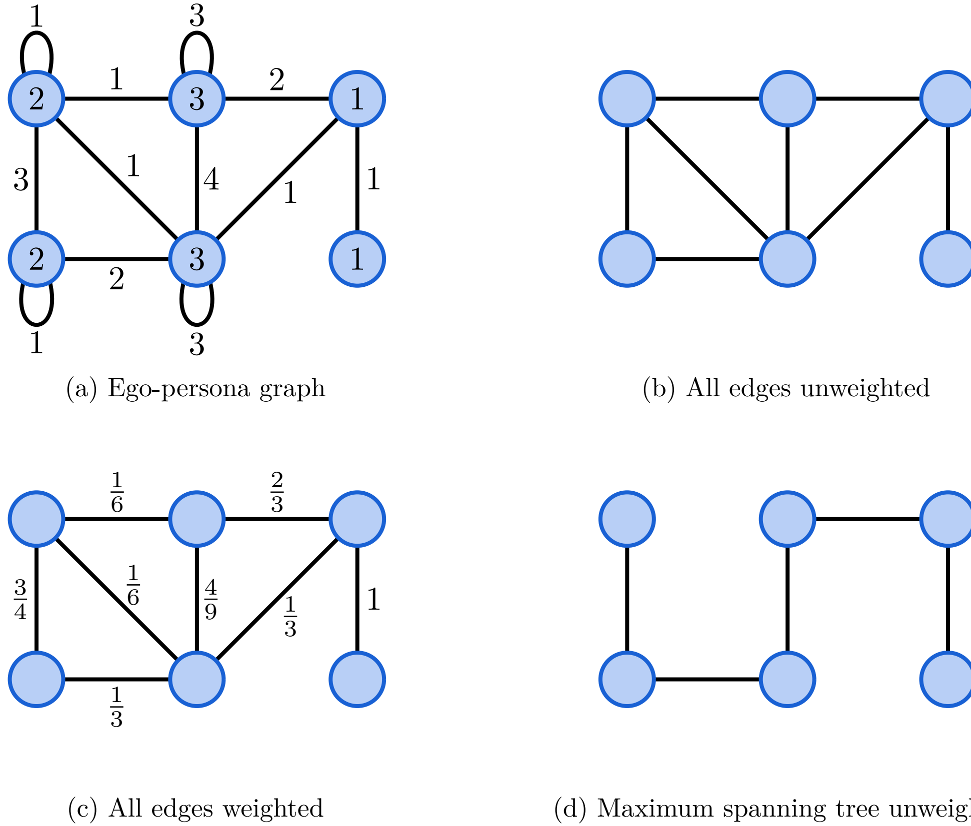
H and the detected clusters are reinterpreted as communities in G. It is obvious that H is equivalent to the persona graph in the ego-splitting framework, i.e. both graphs should only contain non-overlapping communities. When Peacock splits a node u, two personas u_1 and u_2 are created. If a persona is split again, one new persona is created, e.g. u_2 is split into u_2 and u_3. At the end of the first phase, H contains a number of personas for each node in G. The authors acknowledge the problems that follow when the personas are disconnected. The Peacock algorithm places an edge (u_i, u_j) whenever a node is split into two personas u_i and u_j. The authors also propose alternative ways to connect the personas, e.g. by a clique or by connecting all personas with the original node. Because the ego-splitting framework creates all personas at once instead of iteratively, we cannot use the same strategy as the Peacock algorithm. Let P be the graph partition created by the detected clustering, i.e. the reduction of the graph so that each detected cluster corresponds to one node in P. Figure 6.1 depicts the process of creating P from the ego-net clustering. P is a weighted graph and the edges in P are weighted according to the cut between their corresponding clusters. Because we create one persona for each detected cluster, P is identical to the graph of the personas of u. We call P the ego-persona graph of u.

Given a ego-persona graph P, we can also create a modified graph P' that has the same nodes as P but different edge weights. Let u_1 and u_2 be two personas in P. Let $q_1(u_i, u_j) = \frac{w(u_1,u_2)}{|u_1| \cdot |u_2|}$, where $w(u_1, u_2)$ is the weight of the edge (u_1, u_2). q_1 gives us the number edges between the clusters, divided by the possible maximum number of edges $(q_1 \leq 1)$. q_1 is high if the personas u_1 and u_2 are strongly connected. We assign to each edge (u_i, u_j) in P' the weight $q_1(u_i, u_j)$. The advantage of q_1 over the size of the cut is that q_1 gives us a relative strength of the connection between two personas. The size of the cut strongly depends on the number of nodes in the personas/clusters, so large personas tend to have a large cut. In P, the weight of edges between small personas is very low, even if they are strongly connected. There are many ways to normalize the connection strength in the ego-persona graph, e.g. based on additional domain knowledge. We focus on q_1, as it appears to be a straightforward and universally usable normalization of the number of connections.

We connect the personas by inserting additional edges into the persona graph. We propose three strategies to decide which edges are inserted. Figures 6.2a shows a ego-persona graph of a node. Figure 6.2 (b) - (d) depict the strategies 1 - 3:

1. Use all edges in P. For each edge (u_i, u_j) in P, we insert an unweighted edge (u_i, u_j).

2. Use all edges in P'. Let $w_{max} = \max_{i,j} w(u_i, u_j)$ be the maximum edge weight in P'. For each edge (u_i, u_j) in P', we insert a weighted edge (u_i, u_j) with weight $\frac{w(u_i,u_j)}{w_{max}}$, i.e. we normalize the weight so that the maximum edge weight is 1.



(a) Ego-persona graph

(b) All edges unweighted

(c) All edges weighted

(d) Maximum spanning tree unweighted

Figure 6.2: Connect personas of the same node using different strategies. In Figure (a), the numbers inside each node show the size of the corresponding cluster.

3. Use an unweighted maximum spanning tree of P. We apply a maximum spanning tree algorithm to P. For each edge (u_i, u_j) in the spanning tree, we insert an unweighted edge (u_i, u_j).

If two clusters are not connected in the ego-net, we assume that they do not belong to the same community, so we do not need to connect their corresponding personas. In the following, we assume that G is unweighted. For strategies 1 and 3, we only insert unweighted edges into the persona graph. For strategy 2, we insert weighted edges into the originally unweighted persona graph. In this case, we transform the persona graph into a weighted graph by assigning a weight of 1 to each unweighted edge. Using strategy 2 therefore requires that the global clustering algorithm is able to handle weighted graphs. For strategy 3, we use a spanning tree to make sure that all personas are connected to at least one other persona, provided they are not isolated in the ego-persona graph. Intuitively, two clusters are more likely to be in the same community if they are strongly connected. By using a maximum spanning tree, we connect the personas that are most likely in the same community.

By inserting additional edges into the persona graph, we increase the running time of the global clustering algorithm. Depending on the global clustering algorithm, connecting the personas may increase the total complexity of the ego-splitting framework. On major difference between the strategies is the maximum number of inserted edges. Let p be the number of personas of a given node. Strategies 1 and 2 insert up to $\frac{p \cdot (p-1)}{2}$ edges, while strategy 3 inserts at most $p - 1$ edges. Strategies 1 and 2 thus may increase the running time of the global clustering algorithm much more than strategy 3.

Strategy 3 seems like a strong choice, as it provides multiple advantages over the other approaches:

- The number of added edges is linear in the number of personas. Compared to strategies 1 and 2, this should lower the computational complexity for the global clustering algorithm. We still connect the personas that have the best connection in the ego-net. Because we use a maximum spanning tree, each persona is connected to the persona it has the strongest connection to.
- The added edges are unweighted. First, this is good because we keep the persona graph unweighted (assuming that G is unweighted). This allows us to use any global clustering algorithm that works on unweighted graphs, instead of restricting ourselves to clustering algorithms for weighted graphs. Second, each unweighted edge provides a strong connection between two nodes. If we use weighted edges instead, it is possible that an edge has a low weight. Such an edge with a low weight will most likely have no noticeable impact on the global clustering algorithm.

7. Community Clean-Up

We apply the global clustering algorithm to detect clusters in the persona graph. Each persona cluster corresponds to one community in the original graph. A problem is that most of the time, the persona graph is not optimal, because the local clustering algorithm did not produce the optimal results. For example, a node u was split into too many personas. If the global clustering algorithms assigns a different community to each of these personas, then u is assigned to more communities than optimal. If a node is split into too few personas, then global clustering algorithm can only detect too few communities for this node. Many clustering algorithms, e.g. the modularity based Louvain algorithm, aim to create a good partitioning of the graph. A persona with degree one is most likely assigned to the same cluster as its neighbor. Intuitively, we would say that this persona is not part of a community, because it only has a very sparse connection. To improve the strength of the detected community, we want to remove such nodes. If we use a high quality clustering algorithm to detect the communities in the persona graph, it is difficult to improve the detected clusters. If we could improve the clusters with a process, this process could also be included in the clustering algorithm. Instead, we evaluate the communities in the original graph, therefore including information that the clustering algorithm does not have. We assume that the detected communities already have a relatively high quality, and we only add or remove some nodes. This clean-up process is then much faster than creating a complete community from scratch.

We propose to clean up the detected communities using statistical significance. The concept of statistical significance of a community was introduced by Lancichinetti et al. [LRR10] and then used by Lancichinetti et al. [LRRF11] for the overlapping community detection algorithm OSLOM. We also considered other approaches, e.g. based on a stochastic block model [Pei15]. One disadvantage of this approach is that the evaluation is always done on the global level, including all communities. This means that the analysis of a community is affected by the other detected communities. On the other hand, using statistical significance allows us to evaluate each community in isolation, independent of the rest of the cover. We have already presented the basic definitions of statistical significance in Section 5.2, including the r-score, on which we base our clean-up process.

The clean-up process consists of two phases. In the first phase, we analyze each community, removing insignificant nodes and adding significant neighbors. If a community is not significant, we discard it entirely. In the second phase, we look at all discarded communities and try to merge them to create significant communities. We will now describe the two phases in detail.

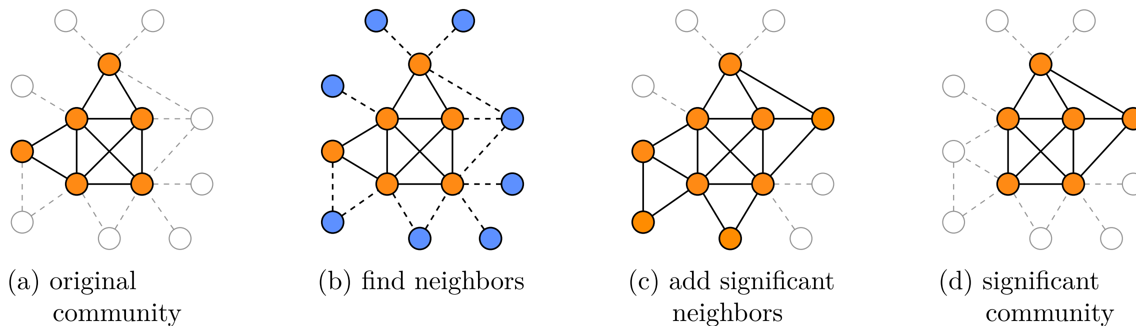


Figure 7.1: The single community analysis. Orange nodes are part of the community, blue nodes are neighbors of the community. The analysis takes a community (a). In phase 1, significant neighbors are added (b-c). In phase 2, insignificant nodes are removed (d). The result is a community containing only significant nodes.

7.1 Single Community Analysis

The community analysis is based on the single cluster analysis of OSLOM. To decide whether a node is significant, we have to define a tolerance P. P is a parameter of the clean-up process, so it is constant for all communities analyses. Let c be the community that we want to analyze.

Lancichinetti et al. make use of order statistics to solve this problem. The r-score is a uniform random variable for nodes of the null model, so it is relatively easy to calculate the ordered statistic distributions. The basic idea is that we compare i with the expected best node according to the null model, i.e. the best expected node (the node with the lowest r-score) is given by the 1st order statistic. Let n_c be the number of nodes in c and $n_e = n - n_c$ be the number of external nodes. The cumulative distribution of the qth order statistic Ω_q is given by

$$\Omega_q(x, n_e) = \sum_{j=q}^{n_e} \binom{n_e}{j} x^j (1-x)^{n_e - j}$$

The core of the analysis consists of making sure that the community only contains significant nodes, i.e. each node $u \in c$ is statistically significant to c. The *Get Significant Nodes* (GSN) routine takes a community c and a set of candidates J and returns all nodes $j \in J$ that are significant to c. Algorithm 7.1 gives the routine in pseudo code. For each node $j \in J$, we calculate its r-score to the community c. Then we sort these candidates ascending by their r-score, giving us the r-scores $r_1, \ldots, r_t$. Let $n_e = |G \setminus c|$ be the number of nodes outside c. Now, starting from $q = 1$, we calculate the value $\phi_q = \Omega_q(r_q, n_e)$. The value ϕ_q gives the probability that the qth best node in the null model would be better (more significant) than the qth best observed node. We calculate ϕ_q for $q = 1, \ldots, t$ until we find a candidate with $\phi_q < P$. In this case, the q examined candidates are significantly better than expected, so they may be part of the community. We continue increasing q until we find a candidate with $\phi_q \geq P$, indicating that the q examined candidates are not significant. However, the previous $q-1$ candidates are significant. The routine then returns these $q-1$ candidates. If there is no candidate with $\phi_q < P$, then all candidates are compatible with the null model. In this case, the GSN routine returns an empty community. The GSN routine takes a community c and a set of nodes J and returns a community $c' = \text{GSN}(c, J)$ that is either statistically significant or empty.

The clean-up procedure of a given community c consists of two phases. Algorithm 7.2 gives the procedure in pseudo code, and we will describe it below. Figure 7.1 depicts how a community is transformed into a significant community.

Algorithm 7.1: GETSIGNIFICANTNODES

Input: Community c, Candidates J
Output: Set of significant candidates c'

```
1  c' ← {}
2  forall j ∈ J do
3      r(j) ← rScore(j)
4  {sort r ascending so that r(1) is the best r-score}
5  q ← 1
6  while q ≤ |J| do
7      s ← Ω_q(r(q), n_e)
8      if s < P then
9          break
10     q ← q + 1
11 if q = |J| + 1 then
12     return
13 while q ≤ |J| do
14     s ← Ω_q(r(q), n_e)
15     if s ≥ P then
16         break
17     q ← q + 1
18 c' ← {q − 1 best candidates}
```

- **Phase 1**: First, we find all neighbors of c, i.e. all nodes outside c that are connected to it. Let N_c be the set of the neighbors of c. We set $J = c \cup N_c$ and execute the GSN routine, yielding a community $c_s = \text{GSN}(c, J)$. If c_s is empty, we have found no significant nodes. This indicates that the community is not significant. However, it is possible that a sub-set of the community is significant, so we try to find such a significant sub-community. Let w be the worst node in c, i.e. the node w with the highest r-score. We remove w from c, giving us a updated community $c' = c \setminus \{w\}$. Now we run the process above again, i.e. we get the neighbors $N_{c'}$ of c' and then compute $c_s = \text{GSN}(c', c' \cup N'_c)$. We remove the worst node and run the GSN routine until the result of the GSN routine is a non-empty community c_s. Alternatively, we stop when we have removed all nodes from the community. At the end, we either have a significant community c_s or an empty community. Note that c_s may contain nodes $u \in N_c$ and thus c_s may be larger than c. If c_s is empty, we consider c not significant and stop the community analysis, returning an empty community. Otherwise, we continue with Phase 2, using c_s.

- **Phase 2**: We repeat phase 1 with c_s. However, we do not consider the neighbors of c_s, i.e. we set $J = c_s$. We run the GSN routine, yielding a community $c'_s = \text{GSN}(c_s, J)$. If the GSN routine returns an empty community, we iteratively remove the worst node of c_s, just like in phase 1. The result of phase 2 is a community c'_s that is either significant or empty.

In phase 1, we add all neighbors that are significant to c, i.e. all nodes that are unexpectedly well-connected to c. In phase 2, we make sure that all nodes in c are significant. This is important because the community may have changed be adding nodes in phase 2. Suppose a node $u \in c$ has l neighbors in c. If we add additional nodes to c, but l stays the same,

Algorithm 7.2: CommunityAnalysis

Input: Community c
Output: Significant community c_s

```
1  c_s ← {}
   // Phase 1
2  while |c| > 0 do
3      N_c ← NeighborsOf (c)
4      J ← c ∪ N_c
5      c_s ← GetSignificantNodes(c, J)
6      if |c_s| > 0 then
7          break
8      w ← WorstNode(c)
9      c ← c \ {w}
   // Phase 2
10 c ← c_s
11 while |c| > 0 do
12     c_s ← GetSignificantNodes(c, c)
13     if |c_s| > 0 then
14         break
15     w ← WorstNode(c)
16     c ← c \ {w}
```

then the r-score of u may decrease. Intuitively, u is not as well-connected to the larger community c. The community c'_s at the end of phase 2 is statistically significant.

The community analysis takes a community c and returns a community c'. If c' is empty, we consider c not significant and discard it. Discarding means that we add it to a set of discarded communities C_D, which we will use in the second phase of the clean-up. If c' is not empty, it is significant, i.e. all nodes outside of c' are compatible with the null model and all nodes inside it are not. We assume that the communities that the ego-splitting framework detects have a good quality, i.e. they only differ from a ground-truth community in a few nodes. However, the community analysis may add a large number of nodes to c and/or remove a large number of nodes from it. The community c' may be completely different than community c, e.g. c' might be much larger or smaller. It is even possible that c and c' are disjoint, i.e. no node of c is part of c'. We define a parameter $\delta, \delta \geq 1$ that limits the change of the community, e.g. $\delta = 2$. If $|c'| < \frac{1}{\delta}|c|$, then c' is shrunk to much. If $|c'| > \delta|c|$, then c' is too big. In both cases, the community was changed drastically by the clean-up process. We do not consider c and c' to be the same community, so we discard c. If we do not discard c, we consider c' a successfully cleaned up community. We add c' to the result cover C'. We repeat the single community analysis for each community in the input cover. As a result, we have a cover C' of (cleaned) significant communities and a set C_D of discarded communities.

7.2 Merge Communities

In the first phase of the clean-up process, we analyze each community, giving us a set of significant communities C' and a set of discarded (not significant) communities C_D. It is possible that by merging two or more of the discarded communities, we are able create a significant community. Intuitively, this may happen if the global clustering algorithm

detects one (ground-truth) community as multiple clusters. It is also possible that one of the discarded communities contains significant sub-communities, as we discarded communities that decrease to much in size in the single community analysis. However, searching for sub-communities is a much more complex problem than testing if merged communities are significant, so we focus on the latter. First, we create a graph H of the discarded communities C_D. Each node in H is one community $c \in C_D$, and the edges in H are weighted according to the number of edges between two discarded communities. We search for communities in H by using a simple non-overlapping community detection algorithm, which we describe in the following. On H, we assign to each node a singleton community. Then we optimize the r-score of the communities, using local moving, i.e. we try to assign each node to a community of its neighbors. For a node $u \in H$ with neighbors v_i, let c_i be the current community of v_i. We calculate the r-score of u to each community c_i of its neighbors. Let r_l be the lowest r-score to a neighbor community c_l, i.e. u is best connected to c_l . We also calculate the r-score r_0 of u to its current community. If $r_l < r_0$, we remove u from its current community and assign it to the community c_l. Each node thus has exactly one community at all times. If $r_l \geq r_0$, we do not change the community assignment of u. We perform this process for each node in H in a random order. We iteratively sweep over all nodes as just described, either until the communities do not change anymore or a maximum number of iterations is reached. Now we have a set of non-overlapping communities on G_u. If a community has a size of one, we ignore it, because it just contains one discarded community. For each community of size one in H, we add the associated discarded communities to a new set of discarded communities C'_D. Let c be a community in H, containing the nodes $u_1, \dots, u_t$. We merge all nodes of the communities u_i in the original graph into one community c', i.e. $c' = \bigcup c_i$, where c_i is the discarded community associated with the node u_i. Then we clean up the community c' using the single community analysis, either yielding a significant community c'' or indicating that c' is not significant. If we get a significant community c'', we add it to the cleaned up cover C'. If c' is not significant, we add it to the new set of discarded communities C'_D. After we have cleaned up all merged communities, we have a set of the remaining discarded communities C'_D. From these communities, we construct a new graph H'. We repeat the entire process described once again on H', trying to find merged significant communities. We add all communities found this way to C' and then terminate the process.

The output of the clean-up process is a cleaned up cover C' that contains only significant communities. We only clean up single communities of C or merge discarded communities of C into new ones, so the number of communities in C' is smaller than or equal to the number of communities in C.

For the most part, our clean-up process is a simplified version of OSLOM. OSLOM can also take a cover as a hint and output a cover of significant communities. However, the entire OSLOM algorithm is much more complex, e.g. OSLOM searches for sub-communities and merges similar communities. If we would use OSLOM to clean up the output of the ego-splitting framework, OSLOM would dominate the running time of the community detection. Our clean-up is less complex, but depends on the fact that the detected communities already have a somewhat good quality, i.e. they are similar to the statistically significant communities.

The clean-up process also solves a fundamental problem of the ego-splitting framework. If the local clustering algorithm did not detect each community in the ego-net of node u as one cluster but instead detected multiple communities as one cluster, then we create too few personas. The persona associated with the cluster of multiple communities can be assigned to only one of its communities by the global clustering algorithm. In the end, u will not be assigned to all of its communities. By using our clean-up, we can assign u to all of its communities. Let c_i be a ground-truth community of u that it is not part of in

the detected cover. Suppose that c_i has been detected with a good quality, i.e. there is a detected community c_i' very similar to c_i. While cleaning up community c_i', we detect that u is strongly connected to c_i' and therefore add u to c_i'. This allows us to assign u to all communities it is significant to, even if there were not enough personas created. The opposite case might also happen, i.e. too many personas are created so u is assigned to too many communities. By cleaning up the communities, we remove u from all of its communities that it is not well-connected to. The clean-up process gives us a strong tool to ensure that each detected community c is actually well-connected, i.e. each node in c is significant to c.

The output of the clean-up process is a cover C'. Each cluster of C' is statistically significant. The clean-up process does not require any additional information provided by the ego-splitting framework, so we can apply the clean-up process to the result of any overlapping community detection algorithm.

8. Experimental Setup

In this chapter, we describe the setup of our experiments. First, we give some details of our implementation of the ego-splitting framework. Then we present the clustering algorithms that we use in our experiments. We consider each algorithm as the local clustering algorithm of the framework, as well as the global clustering algorithm. Next, we present the graphs that we use as the input for the community detection. Finally, we describe the measures we use to evaluate the results of our experiments.

8.1 Implementation Details

We implemented the ego-splitting framework as part of NetworKit [SSM14] in C++. Our implementation is available at `https://github.com/ArminWiebigke/networkit`. The benchmarks are written in Python, using Cython to call the C++ code of NetworKit.

We create the ego-nets using an efficient triangle listing algorithm [OB14]. First, we create a graph $\vec{G} = (V, \vec{E})$ from G. For each undirected edge (u, v) in G, we insert a directed edge from the lower degree node to the higher degree node. If $k_u \leq k_v$, we insert (u, v), else we insert (v, u). To build the ego-net of a node u, we do the following: First, we find all neighbors v_i of u in G and add them to the ego-net $G_u (= V_u, E_u)$. For each neighbor v, we look at all of its outgoing edges (v, w) in $\vec{G}$. If $w \in V_u$, we add the undirected edge (v, w) to the ego-net ($E_u = E_u \cup \{(v, w)\}$). By using $\vec{G}$ instead of G, the worst case complexity is reduced considerably. Lin et al. [LSS12] show that the running time of this algorithm has an upper bound of $4\alpha(G)m$, where $\alpha(G)$ is the arboricity of G. The arboricity of G is the minimum number of spanning forests into which G can be decomposed. A naive algorithm, i.e. the same algorithm executed on G instead of $\vec{G}$, has an running time in $\Theta(k_{max}m)$, where $k_{max} = \max_{u \in V} k_u$ is the maximum node degree in G.

After applying the local clustering algorithm, we store the clustering of the ego-net in a map that maps each node of the ego-net to its cluster. Using a array-like data structure (`std::vector`) is unfeasible, as this would require $O(n)$ memory per ego-net and thus $O(n^2)$ memory for all ego-nets.

Our cover clean-up process based on statistical significance is based on the implementation of OSLOM provided by Lancichinetti [LRRF11]. OSLOM introduces a stochastic element by randomizing the r-score calculation. Each community is analyzed multiple times and is only considered significant if it was considered significant in more than 50% of the iterations. According to the authors, this is important to prevent the identification of significant

communities in a random graph. However, they do not provide a more detailed explanation, and we could not detect any improvement of the detected quality of the communities. We simply use the calculated r-score without randomization because this improves the running time considerably. Likewise, we do not randomize the r-score calculation in the ego-net extension variant Significance, which uses the same code.

We do not create and partition the ego-net in parallel. In theory, the parallelization is trivial as each ego-net can be calculated and processed independently, but Python can not trivially handle this parallelization.

8.2 Algorithms

The ego-splitting framework requires a local clustering algorithm and a global clustering algorithm. A clustering algorithm takes a graph and returns a clustering that assigns at most one community to each node. If the original graph is weighted, the clustering algorithm has to be able to handle weighted graphs. The same holds true for directed graphs. We only evaluate undirected graphs. We present a selection of clustering algorithms with different approaches.

- **PLP** is a simple label propagation algorithm as proposed by Raghavan [RAK07]. PLP is implemented as part of NetworKit.
- **LPPotts** is a label propagation algorithm based on the Absolute Potts Model. We implemented LPPotts as part of NetworKit.
- **PLM** is an implementation of the Louvain algorithm. The Louvain algorithm iteratively optimizes the modularity locally and then aggregates the graph. PLM is implemented as part of NetworKit.
- **LeidenMod**: The Leiden algorithm was proposes by Traag et al. to improve on the Louvain algorithm. The algorithm works similar to Louvain, locally optimizing a fitness function and then aggregating the network. The standard implementation uses modularity as the fitness function. We use an implementation called *leidenalg* that is written in C++ and provides a Python interface. (`https://github.com/vtraag/leidenalg`)
- **Surprise** is a statistical approach to assess the quality of communities [TAD15]. Surprise can be used as a fitness function for the Leiden algorithm. We use the implementation that is part of leidenalg.
- **Infomap** is a clustering algorithm based on the map equation [RAB09]. The algorithms optimizes the description length of a random walk in the graph. We use the implementation provided by the authors (`https://www.mapequation.org/code.html`) which is written in C++ combined with a Python interface.

We compare the ego-splitting framework with other overlapping community detection algorithms. The following algorithms have proven to provide high quality covers [XKS13]:

- **GCE** (Greedy Clique Expansion) greedily expands communities from maximum cliques using a simple fitness function.
- **MOSES** (Model-based Overlapping Seed ExpanSion) is based on an overlapping stochastic block model that assumes that the graph was created by a generative model. MOSES optimizes a derived fitness function to find communities.
- **OSLOM** is based on statistical significance of communities. The algorithm first creates a set of communities an then expands and removes nodes. In the end, each community contains only statistically significant nodes.

See Section 3.2 for a more detailed description of the algorithms.

Parameter	Description	(1)	(2)	(3)
N	number of nodes	2000	2000	2000
k	average degree	$f_\mu(10 \cdot O_m)$	$f_\mu(10 \cdot \overline{O_m})$	$f_\mu(30)$
k_{max}	max degree	$f_\mu(20 + 10 \cdot O_m)$	$f_\mu(20 + 10 \cdot \overline{O_m})$	$f_\mu(50)$
C_{min}	min comm. size	30	30	30
C_{max}	max comm. size	60	60	60
τ_1	degree exponent	2	2	2
τ_2	comm. exponent	2	2	2
μ	mixing factor	0.25	0.25	**0.1-0.8**
O_n	num. overlap nodes	N	**0-2000**	N
O_m	comms per node	**1-7**	2	3

Table 8.1: Parameter sets for the synthetic LFR graphs. The average number of communities per node is given by $\overline{O_m} = (O_m - 1) \cdot \frac{O_n}{N} + 1$.
The average degree is given by $f_\mu(k') = \frac{k'}{1-\mu}$.

8.3 Graphs

Real world graphs are often used to evaluate community detection algorithms. In theory, this makes sense because the main purpose of the algorithms is the application on real-world networks. However, real-world graphs are often very different from synthetic graphs. The ground-truth communities are usually extracted from meta-data of the nodes and not from the structure of the network. There is no guarantee that these ground-truth communities have any structural properties that allows an algorithm to detect them. [HDF14] show that there is in fact a disconnect between structural communities and metadata groups. This means that the desired communities often can not be extracted purely from the structure of the graph, at least with the current modeling of communities. In contrast, synthetic benchmarks like the LFR benchmark [LF09] generate structural communities that are in theory perfectly recoverable. These ground-truth communities provide an important tool to evaluate how an algorithm parameter affects the community detection. We focus our analysis on the synthetic benchmarks, but also provide some evaluations on real-world graphs.

We use synthetic benchmark graphs based on the LFR model. The LFR graph generator takes a set of parameters and generates a graph and a corresponding ground-truth cover. The size of the communities and the degree of the nodes follow a power law distribution, a feature that is also commonly found in real-world networks. Table 8.1 shows the parameters of the LFR graphs that we use in our experiments. We analyze three sets of graphs: Graph set (1) scales the number of communities per node. We increase the average degree accordingly to ensure that each node has a reasonable number of neighbors in each of its communities. Graph set (2) scales the number of overlapping nodes. Each overlapping node has two communities. By increasing the ratio number of overlapping nodes from 0 to 1, we increase the average number of communities from 1 to 2. The average number of communities per node is given by $\overline{O_m} = (O_m - 1) \cdot \frac{O_n}{N} + 1$. For example, for $O_n = 1000$, 50% of the nodes are overlapping, so the average number of communities is $\overline{O_m} = 1.5$. In our evaluations, we often give the results of the graph sets (1) and (2) in a single plot, using the average number of communities per node as the x-axis. Graph set (3) scales the mixing factor, i.e. the ratio of inter-community edges in the graph. For all sets, we use a function f_μ to calculate the average and maximum degree. First, we calculate the number of average intra-community edges $k' = 10 \cdot O_m$, meaning each node has on average 10 neighbors in each of its communities. The average degree is then given by $k = f_\mu(k') = \frac{k'}{1-\mu}$. We use this method to ensure that the average number of neighbors per community of a

	Caltech36	**Smith60**	**Rice31**	**Auburn71**
number of nodes	769	2970	4087	18 448
number of edges	16 656	97 133	184 828	973 918
number of communities	16	44	20	86
largest community	173	627	710	3204
average community size	77.5	117.4	359.6	242.0
average communities per node	1.61	1.74	1.76	1.19

Table 8.2: Properties of the real-world Facebook graphs.

node is independent of the mixing factor. Suppose we set an average degree k and only increase the mixing factor. The mixing factor represents the ratio of inter-community edges that each node has. This means that by increasing the number of inter-community edges, we would at the same time decrease the number of intra-community edges. For example, suppose we have a graph with $om = 3$, $k = 40$ and $\mu = 0.25$. Each node has (on average) $0.25 \cdot k = 10$ inter-community edges and $(40 - 10)/3 = 10$ edges into each of its communities. Now we want to increase mixing factor to $\mu = 0.75$ and keep the average degree the same. Then each node has $0.75 \cdot k = 30$ inter-community edges and $(40 - 30)/3 = 3.\overline{3}$ edges into each of its communities. As each node has fewer neighbors inside its communities, the communities either become weaker connected or smaller. In both cases, the community structure changes drastically. If we instead increase the average degree to $k = f_\mu(30) = 120$, each node has $0.75 \cdot k = 90$ inter-community edges and $(120 - 90)/3 = 10$ edges into each of its communities. The structure of the communities should not change drastically compared to the graph with the lower mixing factor, we just added additional inter-community edges to the graph. The advantage of this method is that we increase the "difficulty" only in one specific aspect. Adding more "noise" to the graphs already makes the community detection harder without changing the communities. One problem of the standard implementation of the LFR model is that each node has the same amount of edges into each of its communities, regardless of the community sizes. This means that large communities are much less dense then small communities. For example, suppose we have two communities of size 20 and 100, the average edges from a node to one of its communities is 10. The community of size 20 has an expected density of $\frac{10 \cdot 20}{19 \cdot 20} = 0.53$, while the community of size 100 only has a density of $\frac{10 \cdot 100}{99 \cdot 100} = 0.1$. We keep the range of community sizes relatively close, between 30 and 60. This results in communities of similar properties, e.g. a similar density inside the community.

We use four real-world networks from a set of Facebook networks [TMP12]. Table 8.2 gives the basic properties of the graphs and their ground truth-communities. Each of these graphs contains the data of the students of one college or university. Each student is represented by a node and an edge in the graphs corresponds to a Facebook friendship between two students. The data sets also include meta-data, such as the gender, major, dorm, year of graduation and others. The meta-data induce the ground-truth communities, e.g. all students from the same dorm form one community. We use only the two attributes dorm and graduation year to create the ground-truth communities, as these attributes have a close relationship with structural communities [TMP12, LC13]. We discard all ground-truth communities with size smaller than five. From the 100 available Facebook graphs, we use four graphs that have shown to contain well detectable communities [HRW17].

8.4 Evaluation Metrics

The ego-splitting framework consists of a number of steps that are executed successively. We attempt to evaluate each of the steps in isolation, because it is difficult to predict how

one change, e.g. the ego-net extension, affects the end result of the framework. We present metrics to evaluate the structure of the ego-net, the result of the local clustering algorithm, and the cover of the graph.

8.4.1 Ego-Net Structure

We start by evaluating the ego-net extension. Applying the local clustering algorithm to the original and the extended ego-net and then comparing the clustering quality is not optimal. Different clustering algorithms might react differently to a change, e.g. one produces a better clustering while the other produces a worse clustering. We define quality measures that are based on solely on the structure of the (extended) ego-net. The ego-splitting framework assumes that the communities of a node form disjoint clusters in the ego-net. Given a ego-node u, its ego-net $G_u = (V_u, E_u)$ and its ground-truth communities $c_1, \dots, c_t$, we focus our analysis on the induced communities c_i^u in the ego-net, $c_i^u = c_i \cap V_u$. In the best case, each induced community c_i^u forms a cluster in the ego-net, i.e. the nodes of c_i^u are densely connected internally and sparsely connected externally. In the original ego-net, this is often not the case. For example, a node belonging to one of the c_i^u might be isolated in the ego-net. In our preliminary evaluations, this actually happened quite often. The local clustering algorithm has obviously no chance to connect that node to its community, as there is just no information in the ego-net that implies this relationship. By extending the ego-net, additional nodes are added, so the clustering algorithm gets more information to work with. In the extreme case, the entire community is included in the extended ego-net. The clustering algorithm should then be capable of detecting that community.

Community Fitness

Let K_c be the sum of the degree of the nodes in c, which can be split into the internal degree K_c^{in} and the external degree K_c^{out}. A community c induces a cut between c and the rest of the graph $G \setminus c$. The conductance of the cut is given by

$$\varphi(c) = \frac{K_c^{out}}{\min(K_c, K_{G \setminus c})} \,.$$

If we assume that $K_c < K_{G \setminus c}$, we can simplify this formula to

$$\varphi'(c) = \frac{K_c^{out}}{K_c} \,.$$

We expect a strong community to have many internal and few external edges, so the conductance of a good community should be small. To get a fitness function that increases for stronger communities, we simply use the complement of the conductance, yielding the fitness function

$$f(c) = 1 - \varphi'(c) = \frac{K_c^{in}}{K_c} \,.$$

Lancichinetti et al. [LFK09] defined a fitness function F that is a generalization of the above formula:

$$F(c) = \frac{K_c^{in}}{{K_c}^{\alpha}}$$

This fitness function is used in the GCE algorithm [LRMH10]. The parameter α is a positive real number that can be tuned. For $\alpha = 1$, $F(c)$ is identical to $f(c)$ of a community. Smaller values of α lead to a better fitness for large communities, while large values of α mean that small communities are preferred. In preliminary benchmarks, we found that if we use $\alpha < 1$, the community fitness is a better indicator for the quality of the local

clustering than $\alpha \geq 1$. This indicates that large communities in the ego-net are easier to detect. We set $\alpha = 0.8$ for the evaluation. We define the community fitness of an ego-net as the average community fitness of all ground-truth communities of the ego-net, i.e. the fitness of the communities we hope to detect.

Coverage

The communities in the ego-net are often only a part of the entire communities, e.g. a global community with 100 nodes has only 20 nodes in the ego-net. For a global community, the fraction of nodes that are part of the ego-net is called the *coverage* [LND16]. Given a community c_i and an induced community c_i^u in the ego-net, the coverage is defined as

$$\operatorname{cov}(c_i) = \frac{|c_i^u|}{|c_i|}.$$

Intuitively, increasing the coverage should improve the strength of the community. If all nodes of a community c_i are inside the ego-net ($\operatorname{cov}(c_i) = 1$), we expect that an optimal clustering algorithm is able to detect that community. The coverage of an ego-net is the average coverage of all ground-truth communities of the ego-node.

External Nodes

In most cases, not all nodes in the ego-net share a community with the ego-node. We refer to these nodes as external nodes. If we could identify all external nodes, we could remove them from the ego-net, most likely making the local clustering easier. External nodes increase the difficulty of the community detection, as they add "noise", i.e. inter-community edges, to the ego-net. While extending the ego-net, it is important to keep the number of added external nodes as low as possible. In the worst case, we add only external nodes, increasing the ratio of inter-community edges and increasing the difficulty of finding the correct communities. In the best case, we add only nodes that share a community with the ego-node. Choosing only non-external nodes for the extension is one of the main goals of the candidate selection algorithm, as we expect external nodes to rarely improve the ego-net structure.

Let n_e be the number of nodes in the original ego-net and n_e^+ be the number of nodes in the extended ego-net. Similarly, let n_x be the number of external nodes in the original ego-net and n_x^+ be the number of external nodes in the extended ego-net. The fraction of added external nodes is given by

$$f_x = \frac{n_x^+ - n_x}{n_e^+ - n_e},$$

and the fraction of external nodes in the extended ego-net is $\frac{n_x^+}{n_e^+}$. A low value of f_x is better, so an optimal ego-net extension would have $f_x = 0$.

8.4.2 Ego-Net Clustering

The ego-net clustering is critical for the success of the ego-splitting framework. It is not immediately clear how a low quality clustering of the ego-net affects the detection of the global communities, as this also depends on the global clustering algorithm. In the best case, each community creates exactly one persona, i.e. each cluster contains (only) all nodes from one community. If we find too few local communities, not enough personas are created, so it impossible for the global clustering algorithm to detect all communities of the node. Also, each persona has only edges to one of the communities (excluding 'random' edges). Each community node that is not in the clustering means that one community-internal edge will not be connected to the persona. This means that the global clustering algorithm has

to detect the community with fewer edges. Let $c_1, \ldots, c_t$ be the ground-truth communities of the ego-node u. Each ground-truth community c_i induces a community $c_i^u = c_i \cap G_u$ in the ego-node. Let $C_u = c_1^u, \ldots, c_t^u$ be the set of induced ground-truth communities. For convenience, we will refer to the c_i^u simply as ground-truth communities. We propose multiple quality measures to evaluate an ego-net clustering D by comparing it to the set of induced ground-truth communities C_u.

Community Segmentation

We define the *community segmentation*, which measures to which degree the communities are split into multiple clusters. For a given ground-truth community $c \in C_u$ and a cluster $d \in D$, we define $p_d(c) = |d \cap c|$ as the number of nodes of c that are in cluster d. The community segmentation

$$\text{seg}(c) = \frac{\max_{d \in D} p_d(c)}{\sum_{d \in D} p_d(c)}$$

gives the fraction of nodes that are not inside the largest cluster the community is split into. For example if, 70% of the nodes of c are assigned to cluster d_1 and 30% of the nodes are assigned to cluster d_2, the community segmentation of c is 0.3. The community segmentation of the entire clustering is the average segmentation of all communities $c \in C_u$. A high community segmentation means that communities are not put into a single cluster. This means that we create too many personas, making the community sparser and thus harder to detect. A clustering algorithm that produces a clustering with a high community segmentation is most likely too restrictive, splitting communities in sub-communities.

Community Merging

We define the *community merging score*, which measures to which degree multiple communities are merged into one cluster. The community merging score of a cluster given by $d \in D$ is

$$\text{merg}(d) = \frac{\max_{c \in C_u} p_d(c)}{\sum_{c \in C_u} p_d(c)} .$$

The score of the entire clustering is the average score of all detected clusters that contain at least one node from the ground-truth communities of the ego-node, i.e. all clusters $d \in D$ with $\text{merg}(d) > 0$. This score describes the average fraction of nodes in each cluster that are not from the ground-truth community with the most nodes in that cluster. For example, given a cluster d, 80% of d are nodes from community c_1 and 20% of d are nodes from community c_2. The community merging score of d is then 0.2. A high community merging score means that multiple communities are detected as a single cluster. In this case, we create too few personas, and the global clustering algorithm can not detect all communities of the ego-node.

Persona Recall

Each ground-truth community should have exactly one persona. If a community is split into many clusters by the local clustering algorithm, many personas are associated with that community, but the connection to each persona is weak. On the other hand, many communities might be put together in one cluster. The associated persona is then part of multiple communities, which is the exact opposite of our goal. So a good clustering algorithm should produce a clustering with two main qualities: First, each persona should only be associated with one community, or at least most of its nodes should belong to one community. Second, each community should have exactly one persona, or at least one persona that is associated with the majority of the nodes.

We propose a measure to evaluate the quality of the created personas. For a given cluster $d \in D$, we define the *dominating* community $c \in C_u$ as the community with the highest number of nodes inside that cluster, i.e. the community that maximizes $p_d(c)$. Intuitively, we assign each persona to the community to which it has the strongest connection. If a community c' does not dominate a cluster, the edges inside that cluster might be lost for the detection of c' in the persona graph, because the associated persona is more likely to be assigned to the dominating community c.

Let D_c be all clusters that are dominated by a community c, i.e. $D_c = \{d \in D | c \text{ dominates } d\}$. We define the persona recall of c as

$$\text{pr}(c) = \max_{d \in D_c} \frac{p_d(c)}{|c|} = \max_{d \in D_c} \frac{|c \cap d|}{|c|}$$

which gives us the recall of the best dominated cluster. The persona recall of the clustering is the average persona recall of all ego-communities.

8.4.3 Cover

Comparing the detected cover with the ground-truth cover is not a trivial task. Multiple measure have been developed to evaluate the similarity of two covers. We use the following two:

F1

Given a detected community c and a ground-truth community c', the F1 Score is defined as the harmonic mean of the *precision* and the *recall*. The precision $P(c, c') = \frac{|c \cap c'|}{|c|}$ is the fraction of nodes of the detected community that are part of the ground-truth community. The recall $R(c, c') = \frac{|c \cap c'|}{|c'|}$ is the fraction of nodes of the ground-truth community that are part of the detected community. Epasto et al. [ELPL17] use the F1 Score to compare two covers. Given a detected cover C and a ground-truth cover C', the F1 Score of the detected cover is given by

$$F_1(C, C') = \frac{1}{|C|} \sum_{c \in C} \max_{c' \in C'} F_1(c, c')$$

For each detected community, the highest F1 Score for all ground-truth communities is calculated. The F1 Score of the cover is the average of all detected communities.

NMI

The NMI (Normalized Mutual Information) is a measure based on information theory. The NMI normalizes the mutual information to the interval [0, 1], 0 meaning that the covers are totally dissimilar and 1 meaning they are identical. Lancichinetti et al. [LFK09] introduced a method to calculate the NMI for two covers. McDaid et al. [MGH11] improved the NMI calculation. We use a implementation provided by McDaid et al. (`https://github.com/aaronmcdaid/Overlapping-NMI`).

9. Experimental Results

We run each benchmark instance ten times and report the average of all runs. When we use LFR graphs, we create a new graph with the given parameters for each iteration. The experiments were run on a server with two 8-Core Intel processor (Xeon Skylake SP Gold 6144 @ 3.50 GHZ) and 192 GB of RAM.

We present results for the ego-splitting phases in the order that the phases are executed in the full algorithm. First, we analyze the ego-net extension. For both extension variants, we evaluate varying values for their parameters and then compare the two extension variants against each other. Then we present results for the ego-net clustering, using various clustering algorithms. Next, we evaluate connecting the personas of each node. We present results for various global clustering algorithms. Then we evaluate the clean-up process. At the end of each section, we conclude which of the tested variants of the ego-splitting algorithm provided the best results. For the following benchmarks, we then use the best (or the two best) variant, e.g. we present results for all global clustering algorithms in combination with the two best local clustering algorithms. Finally, we compare the best configuration of our ego-splitting algorithm with other overlapping community detection algorithms.
We lay the focus of our evaluation on the LFR graphs with a varying number of communities per node, as this graph set makes the graphs harder in an interesting way, i.e. the structure of the graph changes considerably. In contrast, increasing the mixing factor does not change the structure of the communities, and the Facebook graphs have no increasing difficulty. However, we also evaluate the LFR graphs with varying mixing factor and the Facebook whenever they provide interesting results.

9.1 Ego-Net Extension

We evaluate the extension of the ego-net independently of the local clustering algorithm. We consider two strategies to extend the ego-net and test various parameters. First, we present results for the strategy *EdgesScore*, where the fitness of a candidate is given by a simple function that is based on the number of edges between the candidate and the ego-net. Second, we give results for the strategy *Significance*, where the fitness of a candidate is given by its statistical significance to a subset of the ego-net. We report the average of the ego-net metrics over all ego-nets, unless stated otherwise.

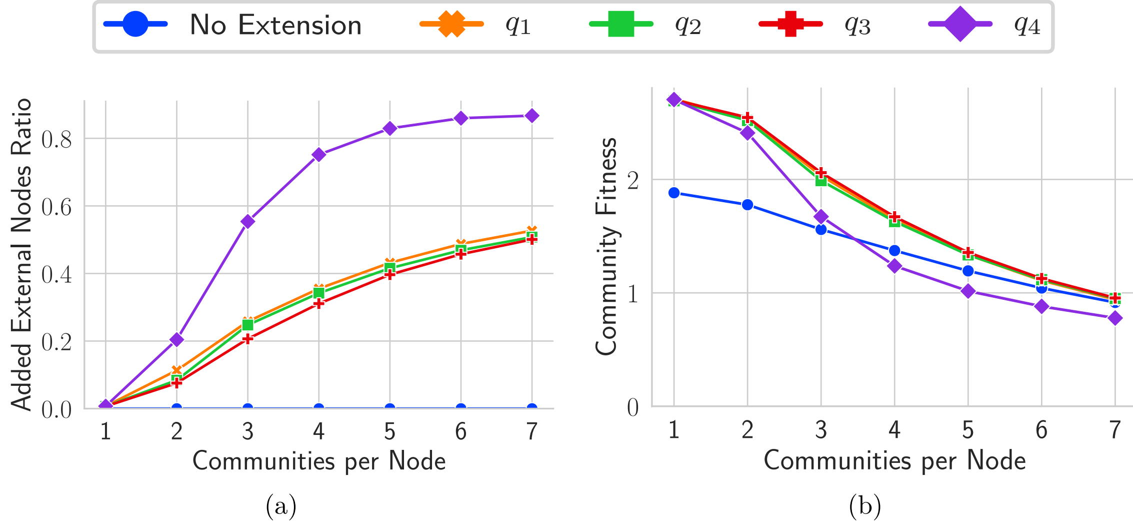


Figure 9.1: Ego-net metrics on LFR graphs with a varying number of communities per node. We compare the original ego-net (blue line) with different strategies to rate the candidates for the ego-net extension using EdgesScore. Shown are the ratio of external nodes among the extended nodes (a) and the community fitness of the ego-net (b).

9.1.1 EdgesScore

We test variations of the extension strategy EdgesScore. First, we compare different fitness functions that are used to rate the candidates. Second, we vary the upper bound of the number of extended nodes. We compare the structural quality of the extended ego-net to the original ego-net. We evaluate the results on LFR graphs with a varying number of communities per node, from 1 up to 7 (graph set (1)).

Candidate Scores

To evaluate the candidates, we define a scoring function q that assigns a score to each candidate. Let k_v^{in} be the number of neighbors of the candidate in the ego-net and k_v be the degree of the candidate v. We consider the following candidate scores:

- $q_1 = k_v^{in}$
- $q_2 = \frac{k_v^{in}}{k_v}$
- $q_3 = \frac{{k_v^{in}}^2}{k_v}$
- $q_4 = \text{rand}(0, 1)$

where rand(0, 1) is a random real value between 0 and 1. Rating the candidates by assigning a random number gives us a baseline for the comparison, where candidates are chosen randomly. Independent of the scoring function, we discard candidates with $k_v^{in} < 3$, as these candidates are unlikely to improve the structure of the ego-net. We set the upper bound for the number of candidates as $o = 5 \cdot {n_e}^{0.5}$ where n_e is the size of the original ego-net. We extend the ego-net using the o candidates with the highest scores. As the tested approaches only differ in the order of the candidates, the number of extended node is the same for all approaches.

Figure 9.1a depicts the ratio of external nodes among the extended nodes, lower values are better. As expected, choosing the candidates at random adds more external nodes than using one of the other scoring functions. For more than four communities per node, the

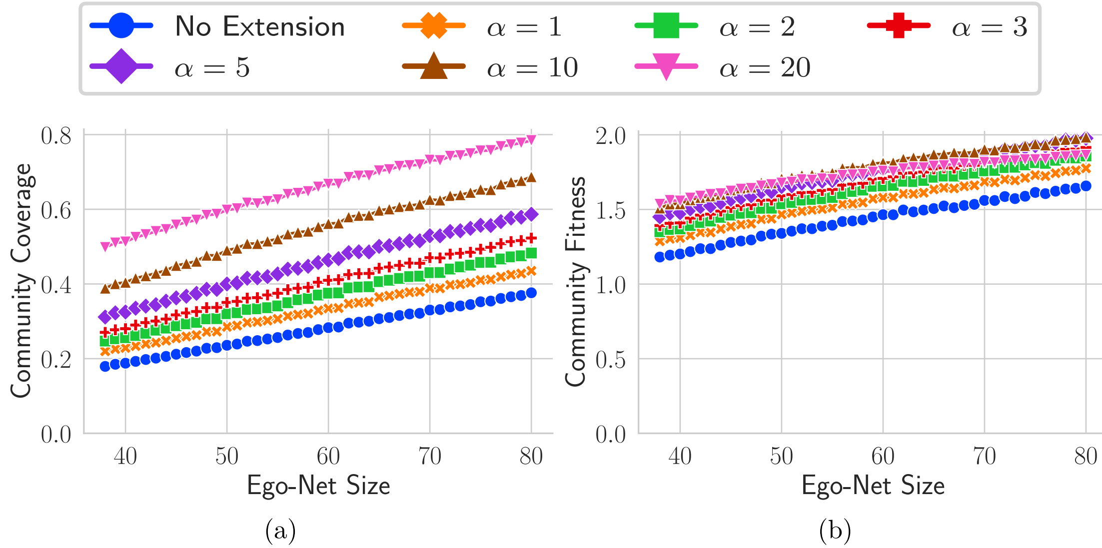


Figure 9.2: Ego-net metrics for different ego-net sizes, using a LFR graph with 4 communities per node (graph set (1)). Shown are the values for the original ego-net (blue line) and varying values of α for the ego-net extension using EdgesScore. Figure (a) depicts the community coverage and Figure (b) depicts the community fitness of the ego-net.

scoring function q_4 adds more than 80% of external nodes, much more than the mixing factor of 25%. Scoring functions q_1, q_2 and q_3 clearly outperform q_4. For all graphs, there is a strict ordering $q_3 < q_2 < q_1$ for the ratio of added external nodes, so q_3 provides the best results. For four communities per node, the scoring function q_3 adds 31% external nodes, q_2 adds 34% and q_1 adds 35% external nodes.
Figure 9.1b depicts the community fitness of the ego-net. As described in Section 8.4.1, the community fitness is the average community fitness F of the ego-communities. Choosing candidates at random improves the community fitness compared to the original ego-net for less than four communities per node. For four or more communities, the extended ego-net with scoring function q_4 has a lower community fitness than the original ego-net. Using the scoring function q_3 provides the best community fitness for all graphs, but the difference to q_1 and q_2 is small. This is expected as q_3 adds the least amount of external nodes to the ego-net. Extending the ego-net using q_3, the community fitness increases by a factor of 1.44 for one community per nodes. As the number of communities per node increases, the advantage of the extension decreases. For six communities per node, the extension increases the community fitness by a factor of 1.08 compared to the original ego-net.

As expected, choosing nodes at random for the extension is not a viable approach. The results for the scoring functions q_1, q_2, and q_3 are relatively similar, but q_3 is strictly better for both metrics on all graphs. We conclude that the scoring function q_3 is the best approach to rate the candidates. For all following benchmarks, we use q_3 as the scoring function.

Maximum Added Candidates

We set the upper bound for the extension as $o = \alpha \cdot {n_e}^{\beta}$, i.e. we add at most o candidates to the ego-net. We choose $\beta = 0.5$ because this ensures that the ratio of extended nodes decreases as the ego-net size increases. It is obvious that the larger the ego-net, the longer is the running time of the local clustering algorithm. Consequently, extending the ego-net increases the running time of the ego-net analysis. If we chose $\beta \geq 1$, the running time would increase even further, especially if the local clustering algorithm does not have a

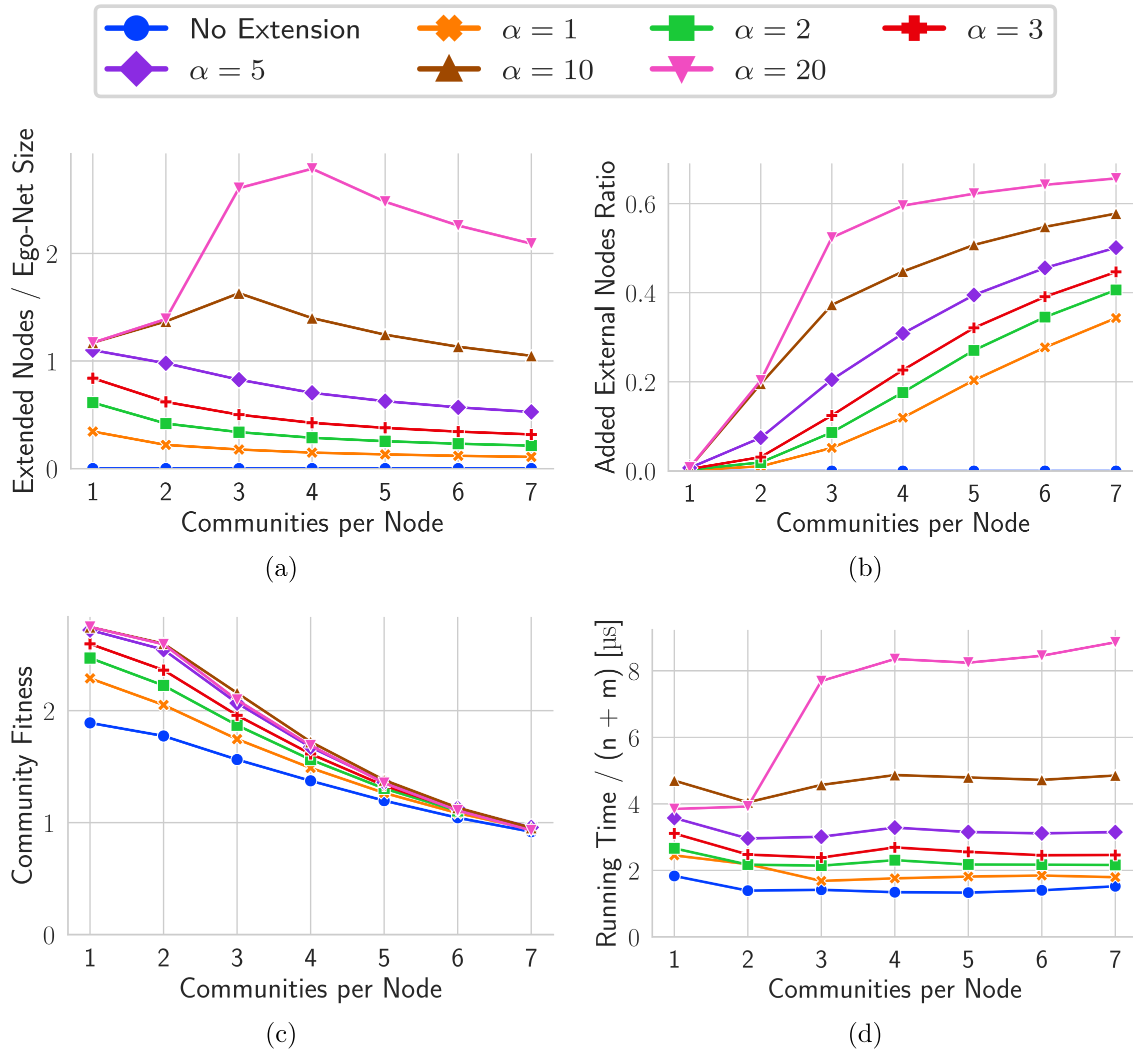


Figure 9.3: Ego-net metrics on LFR graphs with a varying number of communities per node. Shown are the values for the original ego-net (blue line) and varying values of α for the ego-net extension using EdgesScore. The figures depict the community coverage (a), the number of extended external nodes divided by the total number of extended nodes (b), the community fitness of the ego-net (c), and the running time of the local clustering algorithm (PLP) (d).

running time in $O(m + n)$. As we see later, large ego-nets already have a better structural quality than small ego-nets. So even if we extend the ego-net less percentage-wise, the structural quality of the extended ego-net should still be good. We compare values for α from 1 to 20. First, we present results that show the relation between the ego-net size and the community structure for one graph. Second, we analyze the results on graphs with a varying number of communities.

Figure 9.2 shows the community coverage and community fitness on the LFR graph with four communities per node. As described in Section 8.4.1, the community coverage is the fraction of nodes of each ground-truth community of the ego-node that are part of the ego-net. On the x-axis of the plots is the size of the original ego-net, i.e. the degree of the ego-node. We see that the smallest ego-nets have a size of 38 and the largest ego-nets have a size of 80. If we do not extend the ego-net, the coverage increases from 0.18 for ego-nets of size 38 to 0.38 for ego-nets of size 80. The community fitness also increases, from 1.17 for the smallest ego-nets to 1.66 for the largest ego-nets. This means that the larger ego-nets have a better structural quality, i.e. the ego-communities are easier to detect. As expected, the community coverage increases with the number of extended nodes. Compared to the original ego-net, the community fitness increases for all values of α. For $\alpha \leq 5$, increasing the value of α also increases the community fitness for all ego-net sizes. For $\alpha = 5$, the community coverage increases by a factor of 1.72 for ego-net size 38 and by a factor of 1.55 for ego-net size 80. However, increasing the value of α further does not always increase the community fitness. For ego-nets of size 80, the community fitness is identical for $\alpha = 5$ and $\alpha = 10$. For the same ego-net size, $\alpha = 20$ has a lower community fitness than $\alpha = 5$ by a factor of 0.94. The results show that in general, larger ego-nets have a better community coverage and community fitness. Also, choosing $\beta = 0.5$ does not result in large ego-nets having a worse community fitness than smaller ego-nets.

Figure 9.3a shows the ratio of extended nodes compared to the size of the original ego-net. The ratio of extended nodes equals the number of nodes that were added to the size of the original ego-net, e.g. a ratio of 1 means that the size of the ego-net was doubled by the extension. As expected, increasing the value of α increases the number of extended nodes in most cases. An exception are the graphs with less than three communities per node if the value of α is increased from 10 to 20. In this case, no additional nodes are added, as the number of candidates is already smaller than o. As mentioned before, we always discard candidates that have less than three neighbors in the ego-net, so the number of neighbors of neighbors of the ego-node may be higher than the maximum extended nodes.
Figure 9.3b gives the ratio of external nodes among the extended nodes. As the value of α increases, the external ratio also increases. The reason is that candidates that are in a ground-truth community of the ego-node are more likely to have a high score. As we increase the number of extended nodes, we add nodes with a lower score and these nodes are more likely to be external nodes.
Figure 9.3c depicts the community fitness of the ego-net. For $\alpha \leq 5$, increasing the value of α increases the community fitness on all graphs (except for 7 communities per node). Increasing the value of α further increases the community fitness only on some graphs. For $\alpha = 20$, the community fitness is actually equal or lower than for $\alpha = 10$ on all graphs. For the graph with four communities per node, extending the ego-net with $\alpha = 5$ increases the community fitness by a factor of 1.21. Using $\alpha = 10$ instead only increases the factor to 1.25. Compared to $\alpha = 5$, $\alpha = 10$ increases the community fitness only minimally, especially if we consider the fact that double the number of nodes are extended.
Figure 9.3d gives the running time of the local clustering algorithm. For this benchmark, we used a simple label propagation algorithm (PLP). As expected, the additional time caused by the ego-net extension scales approximately linearly with the value of α. For four communities per node, the running time for $\alpha = 10$ is slower than the running time

for $\alpha = 5$ by a factor of 1.55. If a more complex algorithm is used as the local clustering algorithm, we expect that the running time increases even faster for increasing values of α.

In conclusion, increasing the value of α increases the community fitness only up to maximum factor, while the running time scales at least linear with α. For all following benchmarks, we set $\alpha = 5$, as this value provides a good trade-off between the community fitness on the one hand and the running time on the other hand.

9.1.2 Significance

In this section, we present results of the ego-net extension using the extension strategy *Significance*, which evaluates candidates based on their statistical significance. We set the upper bound for the extension as $o = 5 \cdot {n_e}^{0.5}$ for all benchmarks, the same as for EdgesScore. In practice, this upper bound is rarely reached, so we do not test higher upper bounds. To decide if a candidate is significant, we use the local clustering algorithm to partition the original ego-net and then calculate the significance of the candidates to each of the detected clusters. The extension is therefore not independent of the clustering algorithm. We present results using LeidenMod as the local clustering algorithm, as it provides a good quality. See Section 9.2 for detailed results. We also tested different algorithms, but the optimal parameters are similar for all algorithms.
To facilitate a well-structured analysis of the various variants and parameters of Significance, we use the method described below. We start with the most basic form of the Significance extension procedure that is described as follows: We apply the local clustering algorithm to the ego-net. For each candidate, we evaluate its significance to all detected clusters. A candidate is added if there is at least one cluster to which the candidate is significant. We do not add candidates iteratively, and we do not iteratively extend and partition the ego-net. In the following sections, we compare variants of the extension as described in Chapter 5. At the end of each section, we decide which variant (or parameter value) performs best and incorporate it in the "current best" extension algorithm. We then use the current best variant as the base algorithm for all following benchmarks. By using this method, we seek to isolate the evaluation of one parameter as good as possible. First, we present results for checking the significance of the candidates against merged clusters. Then we test if only looking at a maximum number of candidates is sufficient for a good result. Next, we extend the ego-net iteratively by updating the remaining candidates. After that, we show whether only recalculating the significance of the updated candidates decreases the quality of the extension. Finally, the extension is done iteratively, taking the result of the clustering algorithm on the extended ego-net as the basis for the candidate evaluation.

Merge Groups

To test the significance of a candidate, we evaluate its significance to each cluster that was detected by the local clustering algorithm. Additionally, we can merge multiple clusters and evaluate the significance of the candidate to these merged clusters. In the following, we compare the variant that only checks the detected clusters (*Single*) with the variant that also checks the merged clusters (*Merged*).

Figure 9.4a depicts the ratio of extended nodes, i.e. the number of extended nodes divided by the size of the original ego-net. Considering the merged clusters increases the ratio of extended nodes for all graphs. For the graph with one community per node, the variant Merged increases the number of extended nodes by a factor of 2.05. As the number of communities per node increases, the ratio of extended nodes decreases for both variants. For six communities per node, the variant Single extends by a factor of 0.014, while the variant Merge extends by a factor of 0.045, so Merge increases the number of extended nodes by a factor of 3.2. Figure 9.4b shows the ratio of added external nodes, i.e. the

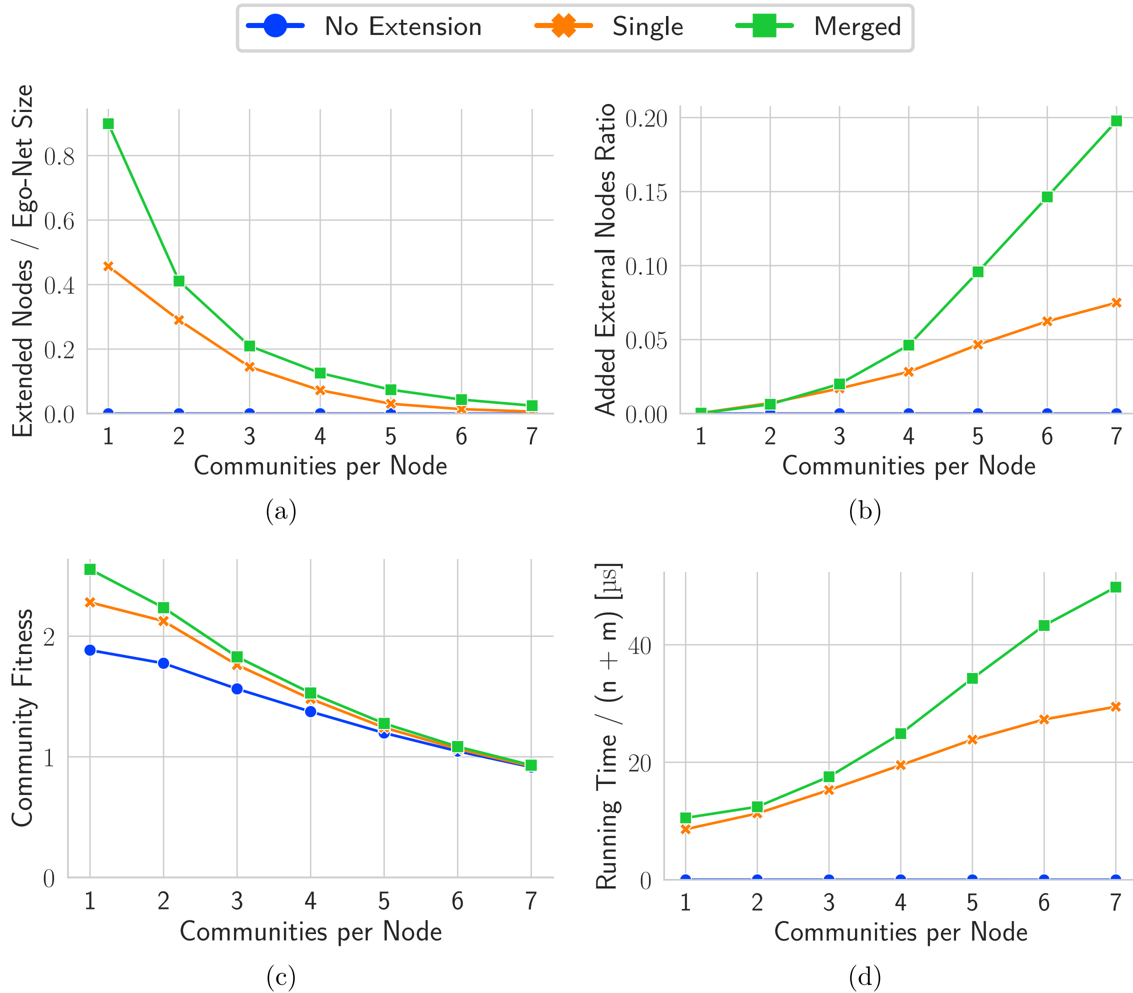


Figure 9.4: Ego-net metrics on LFR graphs with a varying number of communities per node. The ego-net is extended with Significance, checking the significance of the candidates either only against the detected clusters (Single) or also against merged clusters (Merged). Shown are the number of extended nodes divided by the size of the original ego-net (a), the number of extended external nodes divided by the total number of extended nodes (b), the community fitness of the ego-net (c), and the normalized running time of the extension process (d).

number of extended external nodes divided by the total number of extended nodes. The variant Merge increases the ratio of external nodes for all graphs compared to Single. For seven communities per node, the variant Single has a ratio of external nodes of 0.082, while the variant Merge has a ratio of 0.204. This is an increase by a factor of 2.49. However, even the variant Merge adds at most 20% external nodes, so the quality of the added nodes is still high.
Figure 9.4c gives the community fitness of the ego-net. For all graphs, the variant Merge has a higher community fitness than the variant Single. Both variants result in an extended ego-net with a higher community fitness than the original ego-net. The variant Merge is better than the variant Single by a factor of up to 1.12 (one community per node).
Figure 9.4d shows the normalized running time of the extension process. The variant Merge increases the normalized running time for all graphs, up to a factor of 1.73 for seven communities per node.

In conclusion, the variant Merge adds additional nodes to the ego-net and increases the community fitness. On the other hand, the running time increases, but never by a factor of

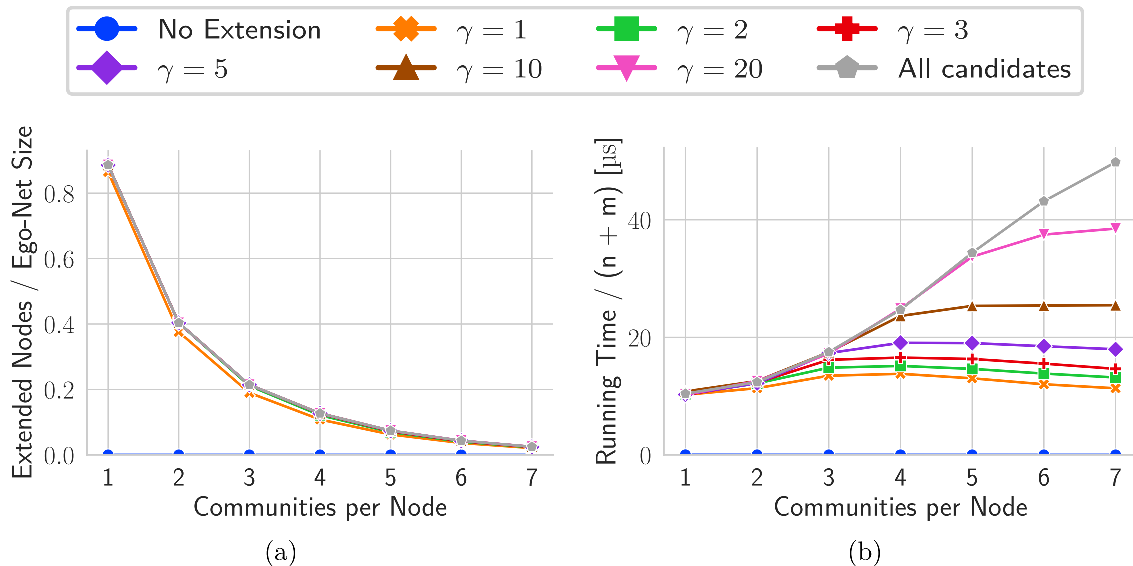


Figure 9.5: Ego-net metrics on LFR graphs with a varying number of communities per node. The ego-net is extended with Significance, using varying values of γ or evaluating all candidates (gray line). Shown are the number of extended nodes divided by the size of the original ego-net (a), and the normalized running time of the extension process (b).

two or greater. We conclude that the variant Merge is the better one, because it considerably increases the number of extended nodes, so we use it for all following benchmarks.

Check Max Candidates

To improve the running time of the extension process, we limit the number of evaluated candidates. We set an upper limit $\bar{o} = \gamma \cdot o, \gamma \leq 1$, where o is the upper bound for the number of added nodes as described earlier ($o = 0.5 \cdot {n_e}^{0.5}$). We sort the candidates by the number of neighbors they have in the ego-net and only evaluate the $\bar{o}$ candidates with the most neighbors. In the following, we compare varying values of γ, ranging from 1 to 20.

Figure 9.5a depicts the ratio of extended nodes, i.e. the number of extended nodes divided by the size of the original ego-net. In general, higher values of γ result in more candidates begin extended. This is expected, as we evaluate more candidates that could potentially be significant. If we evaluate all candidates, the number of extended nodes compared to $\gamma = 1$ increases by a factor of up to 1.2 (for five communities per node). For higher values of γ, the difference becomes very small. If we set $\gamma = 10$, the number of extended node is identical to the number of extended nodes if we evaluate all candidates.
Figure 9.5b shows the running time of the extension process, normalized by the number of nodes and edges. As expected, setting higher values of γ increases the normalized running time, as we have to evaluate (superlineraly) more candidates. If we evaluate all candidates, the normalized running time increases as the number of communities per node increases. This is expected, because the average node degree increases linearly, so the average number of candidates (neighbors of neighbors) increases quadratic with the number of communities per node. If we instead only evaluate $\bar{o}$ candidates, the normalized running time does not increase beyond a given point. For $\gamma = 10$, the normalized running time for four communities per node is 23 µs. The normalized running time does not exceed 25 µs, even for seven communities per node. As intended, limiting the number of evaluated candidates limits the maximum time we spend for the candidate evaluation. Using $\gamma = 10$ increases the running time compared to $\gamma = 1$ by a factor of at most 2.2.

The results show that setting an upper limit $\bar{o}$ improves the running time on complex graphs. Using $\gamma = 1$ reduces the running time drastically, but it also reduces the number of added nodes. We conclude that $\gamma = 10$ is a good choice, as this should ensure that we evaluate practically all significant candidates, while the running time is still acceptable in the worst case. For the following benchmarks, we set $\gamma = 10$.

Extend Iterative

After finding all significant candidates, we add them to the ego-net, which changes the structure of the ego-net. We assign each added candidate to the group that is has the highest significance to. Then we update all remaining candidates, adding additional edges if they are connected to a candidate we just added. Note that we do not consider new candidates. This process is repeated for at most I_{max} iterations. If no candidates were added in the last iteration, the iterative process stops. The parameter thus only provides an upper limit for the number of iterations. We test values of I_{max} between 0 and 100. Zero iterations means that the significance is only checked once and no candidates are updated.

Figure 9.6 depicts the ego-net metrics for varying values of I_{max} on the LFR graphs with varying numbers of communities per node. Figure 9.6a shows the number of extended node divided by the size of the original ego-net. For higher numbers of iterations, the number of extended nodes increases. However, using more than ten iterations does not increase the number of added nodes any further. Compared to using $I_{max} = 0$, using $I_{max} = 10$ increases the number of extended nodes by a factor of 2.5 for four nodes per community. Figure 9.4b shows the ratio of extended external nodes. The ratio is similar for all values of I_{max}, which means that the quality of the additionally extended candidates is similar to the quality of candidates added in the first iteration. Figure 9.4c depicts the community fitness of the ego-net. The community fitness increases with additional iterations, but only minimally for more than three iterations. This is expected as more nodes are extended, but the quality of the extended nodes does not decrease. For three nodes per community, using $I_{max} = 10$ increases the community fitness by a factor of 1.1 compared to $I_{max} = 1$. Figure 9.6d gives the running time of the extension process, normalized by the number of nodes and edges of the graph. As expected, the running time increases if we increase I_{max}. However, for $I_{max} > 10$ the running time does not increase any further. We can assume that after about ten iterations, there is only a very small chance to find additional significant candidates, so it is rare that more than ten iterations are executed.

In conclusion, iteratively adding candidates increases the number of extended nodes and also increases the community fitness considerably. However, for values of I_{max} larger than ten, the number of extended nodes does not change any further. For the following benchmarks, we set $I_{max} = 10$ to ensure that we add as many nodes as possible to the ego-net.

Check Only Updated Candidates

After we add nodes to the ego-net, we iteratively update the remaining candidates and recalculate the significance of all candidates. To improve the running time, we can reevaluate only the candidates that have connections to the candidates that we added in the last iteration. We use the term *improved* candidates, because these candidates are more likely to have gained a higher significance. We compare two variants, either reevaluating all candidates (All) or only reevaluating improved candidates (Only Improved).

Figure 9.7a shows the number of extended node, divided by the size of the original ego-net. Reevaluating only the improved candidates decreases the number of extended nodes on some graphs. Evaluating all candidates increase the number of extended nodes by a factor

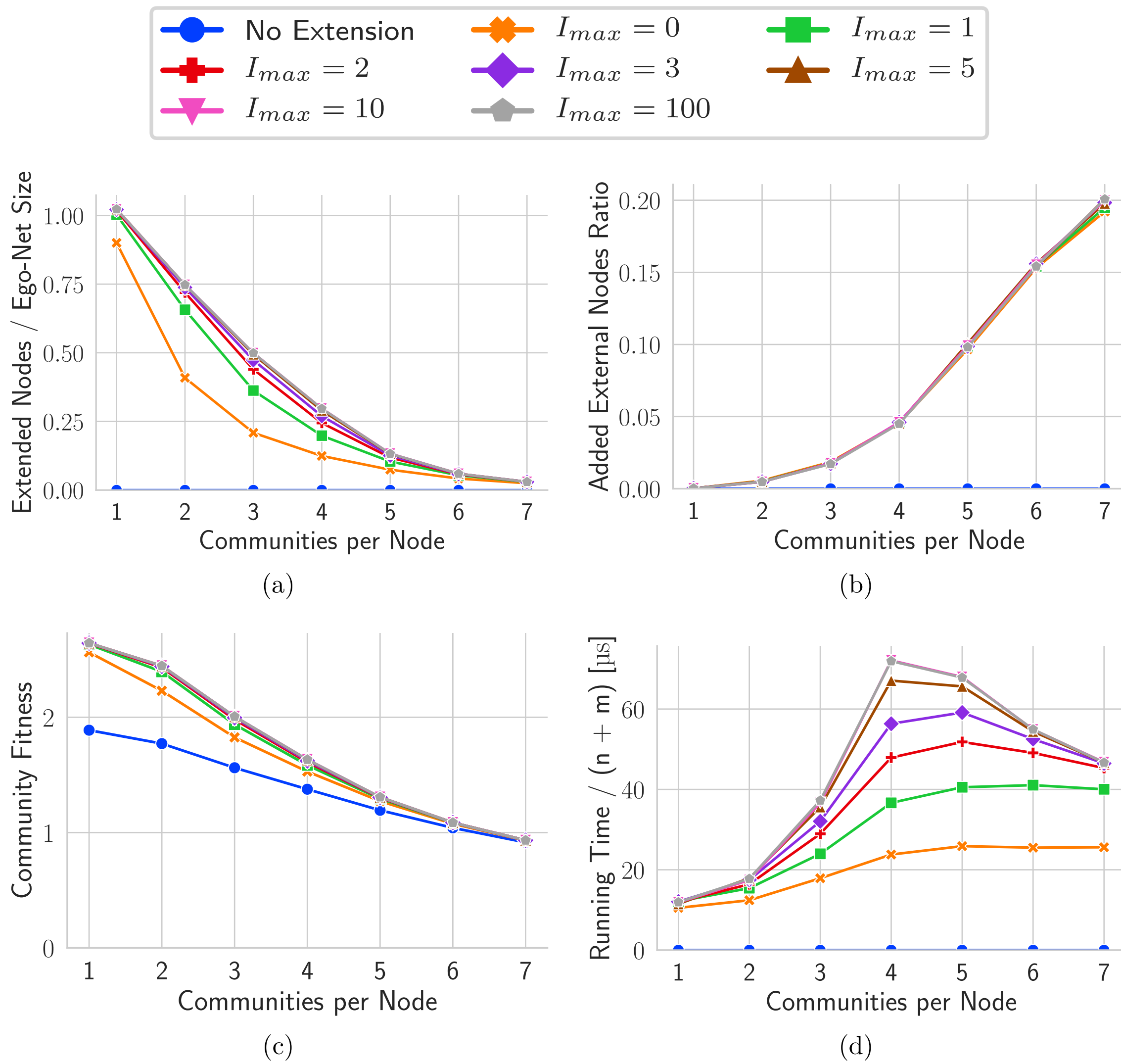


Figure 9.6: Ego-net metrics on LFR graphs with a varying number of communities per node. The ego-net is extended with Significance, updating and reevaluating candidates for a varying (maximum) number of iterations I_{max}. Shown are the number of extended nodes divided by the size of the original ego-net (a), the number of extended external nodes divided by the total number of extended nodes (b), the community fitness of ego-net (c), and the normalized running time of the extension process (d).

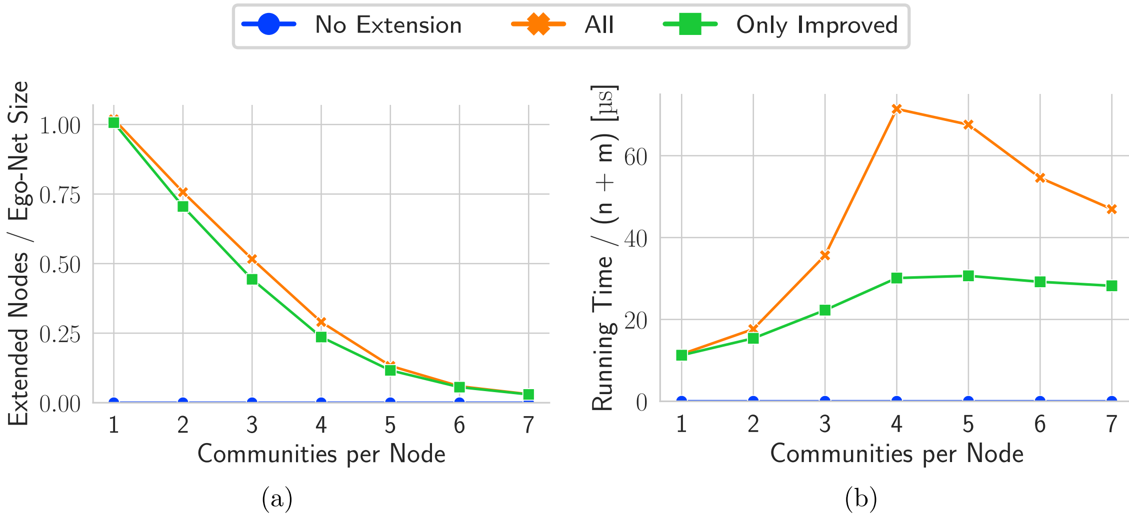


Figure 9.7: Ego-net metrics on LFR graphs with a varying number of communities per node. The ego-net is extended with Significance, either reevaluating all candidates or only the improved candidates. Shown are the number of extended nodes divided by the size of the original ego-net (a), and the running time of the extension process (b).

of up to 1.23 (four communities per node) compared to evaluating only the improved candidates. This is relatively unexpected. When we add nodes to the ego-net, the values used to calculate the significance change, e.g. the number of external nodes and stubs. In general, theses changes should not result in a higher significance of a candidate j if j did not gain additional connection. However, it seems that sometimes such a candidate has a higher significance after the update.
Figure 9.7b depicts the running time of the extension process, normalized by the number of nodes and edges of the graph. As expected, reevaluating only the improved candidates improves the performance. Evaluating all candidates is considerably slower for three or more communities per node, up to a factor of 2.25 for four communities per node.
We conclude that reevaluating only the improved candidates leads to the expected improvement of the running time, but with slightly less extended nodes. We consider this a worthy trade-off, so in the following experiments, we only evaluate the improved candidates.

Cluster-Extend Iteratively

We start the extension process by clustering the original ego-net, then we calculate the significance of the candidates to the detected clusters. After we extend the ego-net, we apply the local clustering algorithm on the extended ego-net. We can repeat this process iteratively, taking the new clusters as the base to calculate the significance of the candidates. We expect that the detected clusters on the extended ego-net have a higher quality than the clusters detected on the original ego-net. If this is the case, we expect that it becomes easier to detect good candidates. After each iteration, we have to evaluate all candidates again, as a candidate that was significant to the previous clustering could be not significant to the current clustering. Let I_c be the number of iterations. $I_c = 1$ means that we only cluster the original ego-net and extend with Significance once. We compare values of I_c from 1 to 8.

Figure 9.7a depicts the number of extended node divided by the size of the original ego-net. The number of extended nodes increases for larger values of I_c. As expected, applying the local clustering algorithm to the extended ego-net results in a better clustering and thus makes it easier to detect significant candidates. For four communities per node, $I_c = 2$

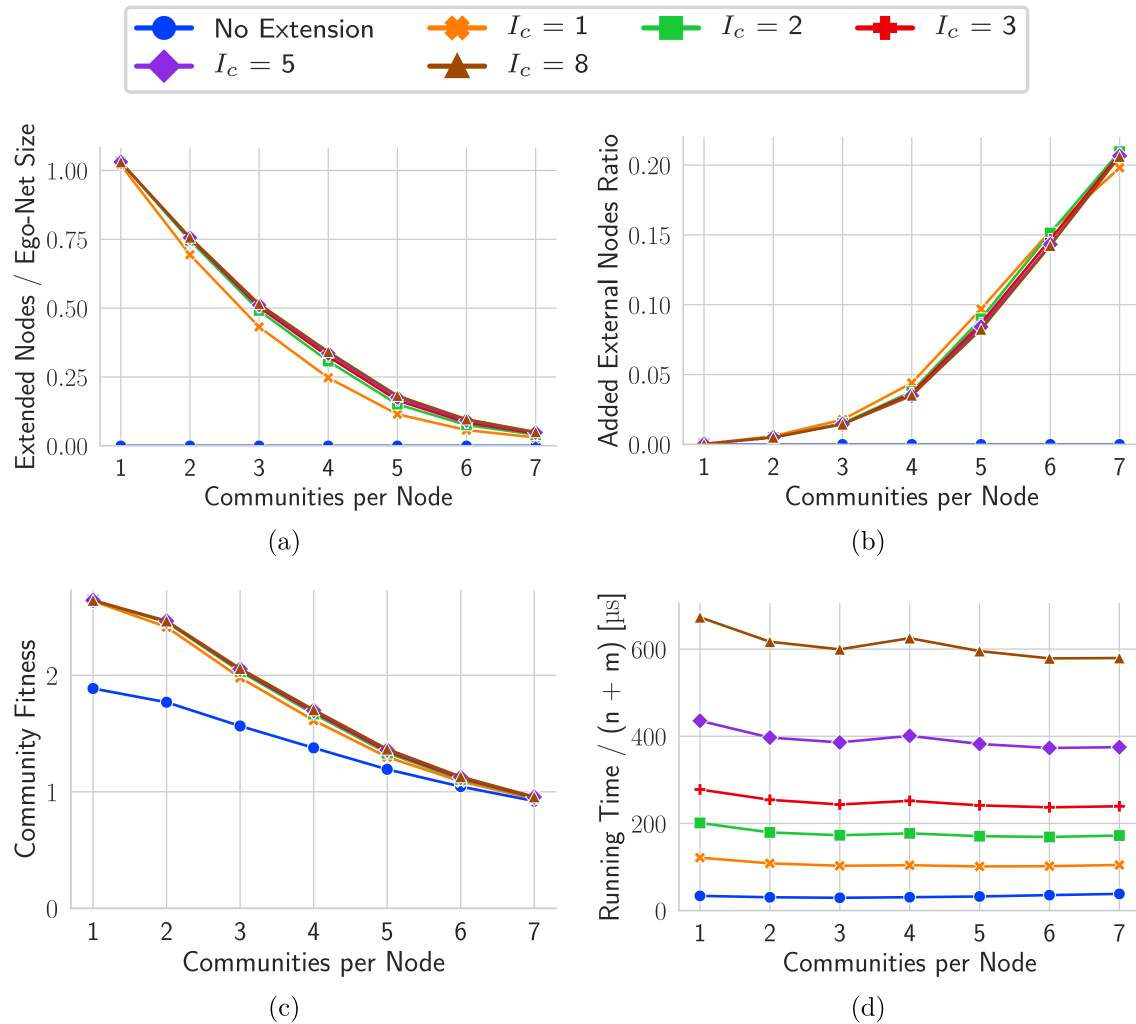


Figure 9.8: Ego-net metrics on LFR graphs with a varying number of communities per node. The ego-net is extended with Significance, extending and clustering for a varying number of iterations I_c. Shown are the number of extended nodes divided by the size of the original ego-net (a), the number of extended external nodes divided by the total number of extended nodes (b), the community fitness of the ego-net (c), and the normalized running time of the extension and clustering process (d).

increases the number of extended nodes by a factor of 1.25 compared to $I_c = 1$, while $I_c = 3$ increases the number of extended nodes only by a factor of 1.05 compared to $I_c = 2$. The number of extended nodes increases the most when we cluster the extended ego-net the first time ($I_c = 2$). With each additional iteration, the number of extended nodes only increases by a small amount. The difference in the result between $I_c = 5$ and $I_c = 8$ is minimal.

Figure 9.8b shows the ratio of extended external nodes. For $I_c > 1$, all variants have similar ratios of extended nodes. Using $I_c = 1$ adds more external nodes for less than six communities per node, up to a factor of 1.2. For seven communities, using more than one iteration adds more external nodes by a factor of up to 1.12.

Figure 9.8c shows the community fitness of the ego-net. With increasing values of I_c, the community fitness increases for all graphs. For four communities per node, $I_c = 3$ increases the community fitness by a factor of 1.05 compared to $I_c = 1$.

Figure 9.8d shows the running time of the extension process plus the running time of the

local clustering algorithm, normalized by the number of nodes and edges of the graph. As expected, the running time scales linearly with the value of I_c. Increasing the value of I_c from 1 to 2 approximately doubles the running time, because both the extension process and the clustering algorithm are executed twice.

In conclusion, clustering iteratively based on the extended ego-net increases the quality of the ego-net extension. However, after more than three iterations, the improvement is minimal. Therefore, $I_c = 3$ is a good choice, and we use it for the following benchmarks.

9.1.3 Original vs. Extended Ego-Net

In the previous sections, we have optimized the strategies EdgesScore and Significance to extend the ego-net. Now we compare the two strategies and analyze the differences of the extended ego-nets to the original ego-net.

Figure 9.9 shows the results on the LFR graphs with varying numbers of communities per node. Figure 9.9a gives the ratio of extended nodes. Using EdgesScore extends the ego-net much more than Significance, especially if the graph has many communities per node. For four communities per node, EdgesScore extends more nodes than Significance by a factor of 2.3. For seven communities per node, this factor increases to about 12. For the harder graphs, Significance is not able to identify many significant nodes.
Figure 9.9b gives the ratio of extended external nodes. Extending with EdgesScore adds many more external nodes to the ego-net. For four communities per node, Significance has a ratio of added external nodes of 0.035. For the same graph, EdgesScore has a ratio of 0.309, a factor of 8.8 higher than Significance. For seven communities per node, EdgesScore has a higher ratio than Significance by a factor of 2.4. As expected, statistical significance is a good indicator for candidates of high quality.
Figure 9.9c depicts the coverage of the ground-truth communities of the ego-node. For less than four communities per node, both variants of the ego-net extension increase the coverage considerably. For two communities per node, EdgesScore increase the coverage by a factor of 2.16, while Significance increases the coverage by a factor of 2.05. As the number of communities increases, the coverage for both extension variants decreases. However, the coverage for Significance decreases more than for EdgesScore. For seven communities per node, EdgesScore increases the coverage by a factor of 1.37, while Significance increases the coverage by only a factor of 1.06. The reason is that on this graph, Significance extends only a few nodes, so there are not enough nodes to considerably increase the coverage.
Figure 9.9d shows the community fitness of the ego-net. For one community per node, using EdgesScore increases the fitness by a factor of 1.43 and using Significance increases the fitness by a factor of 1.40. For more than two communities per node, the community fitness for both extension variants is nearly identical. The difference between the fitness of the original ego-net and the fitness of the extended ego-net decreases for an increasing number of communities per node. For seven communities per node, extending the ego-net increases the fitness by a factor of 1.04.
We have seen that EdgesScore extends many more nodes than Significance for the harder synthetic graphs. At the same time, EdgesScore adds many external nodes on these graphs. In contrast, Significance adds a low amount of external nodes. The statistical significance is successful in identifying the extension candidates that belong to the ground-truth communities.

Figure 9.10 shows the results on the Facebook graphs. EdgesScore extends more nodes than Significance, up to a factor of 1.21 for the graph Caltech36. For the other three graphs however, the difference is at most a factor of 1.04. The ratio of added external nodes is similar to all graphs. This is clearly different than on the LFR graphs, where Significance often adds a lot less external nodes. It seems that Significance is not able to detect high

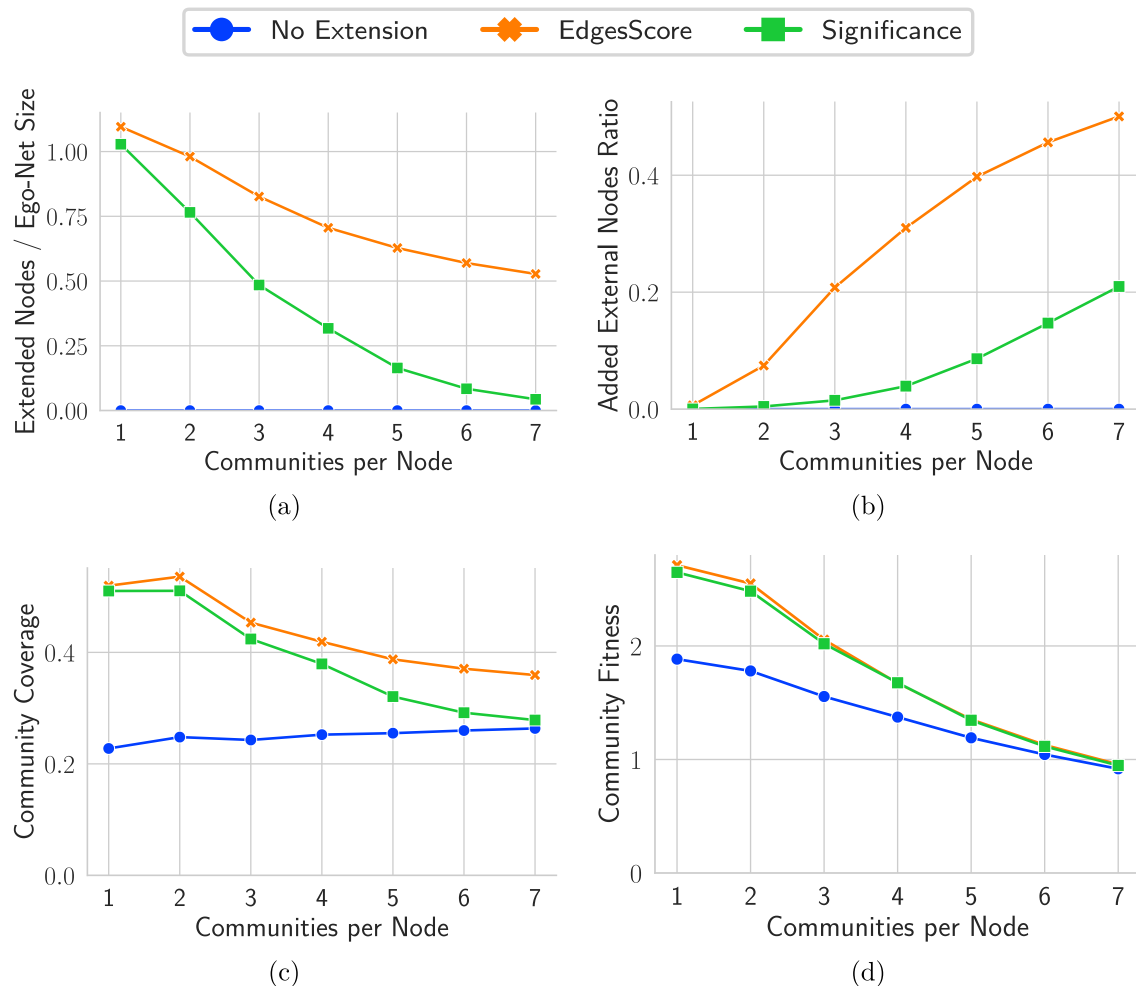


Figure 9.9: Ego-net metrics on LFR graphs with a varying number of communities per node. The ego-net is extended using either EdgesScore or Significance. Shown are the number of extended nodes divided by the size of the original ego-net (a), the number of extended external nodes divided by the total number of extended nodes (b), the community coverage (c), and the community fitness (d).

quality candidates on the Facebook graphs. Both extension variants result in a similar community coverage and community fitness. The community fitness increases by a factor of up to 1.59 for the graph Caltech36. In contrast, the fitness and coverage of the original ego-net is identical to the extended ego-net on the graph Auburn71. On Auburn71, the coverage has a value of 0.91, while the highest coverage on the LFR graphs is 0.52. At the same time, the community fitness on Auburn71 is much lower, with a value of 0.2 compared to values between 0.92 and 2.7 on the LFR graphs. The ego-net extension increases the coverage on the Facebook graphs at most by a factor of 1.22. As we see from the results, the community structure is quite different on the different Facebook graphs. Extending the ego-net improves the structural quality only on some Facebook graphs.

Figure 9.11 gives the running times of the extension process plus the running time of the local clustering algorithm, divided by the number of nodes and edges. Because we apply the local clustering algorithm when extending with Significance, the running time is much higher compared to EdgesScore. Note that the normalized running time is nearly constant for all graphs. Extending the ego-net with EdgesScore increases the running time by a factor of about 2.3 compared to using no extension. Extending with Significance increases

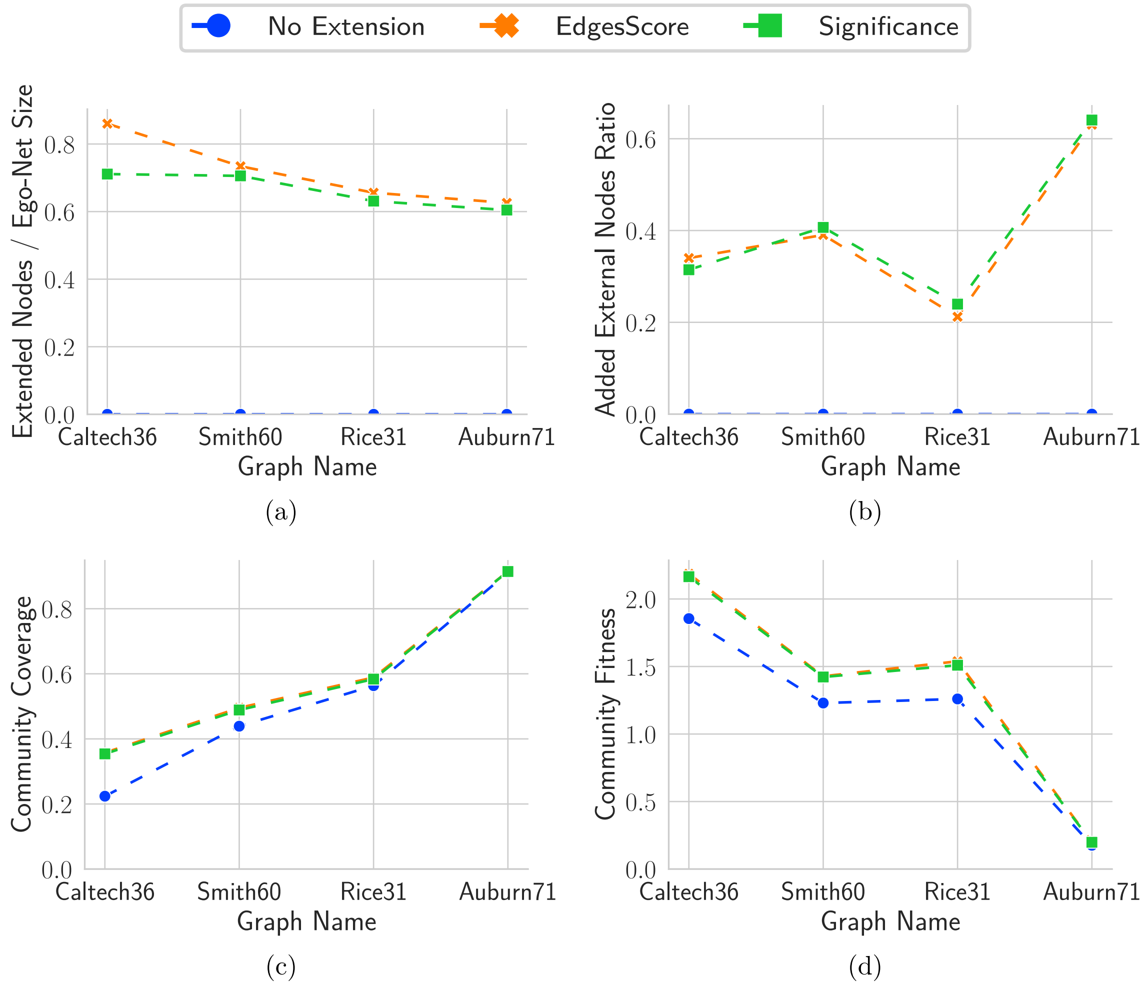


Figure 9.10: Ego-net metrics on the Facebook graphs. The ego-net is extended using either EdgesScore or Significance. Shown are the number of extended nodes divided by the size of the original ego-net (a), the number of extended external nodes divided by the total number of extended nodes (b), the community coverage (c), and the community fitness (d).

the running time by a factor of 8 to 9. On the Facebook graphs, the results are quite similar. Compared to the running time with the original ego-net, using EdgesScore increases the running time by a factor of 2 and using Significance increases the running time by a factor of 7 to 8.

In conclusion, EdgesScore extends many candidates with a low quality on the harder LFR graphs, while Significance extends few high quality nodes. The disadvantage of Significance is that it does find only very few significant candidates if the ego-net is difficulty to cluster, e.g. the number of communities per node is high. On the real-world graphs, EdgesScore and Significance behave much more similarly. For both the synthetic and real-world graphs, the community fitness is nearly identical for the extension variants. The extension consistently increases both the community coverage and the community fitness, except on one Facebook graph. The results do not show that one of the two extensions variant is clearly superior to the other one. We evaluate the results of the local clustering algorithms for both extension variants in the next section.

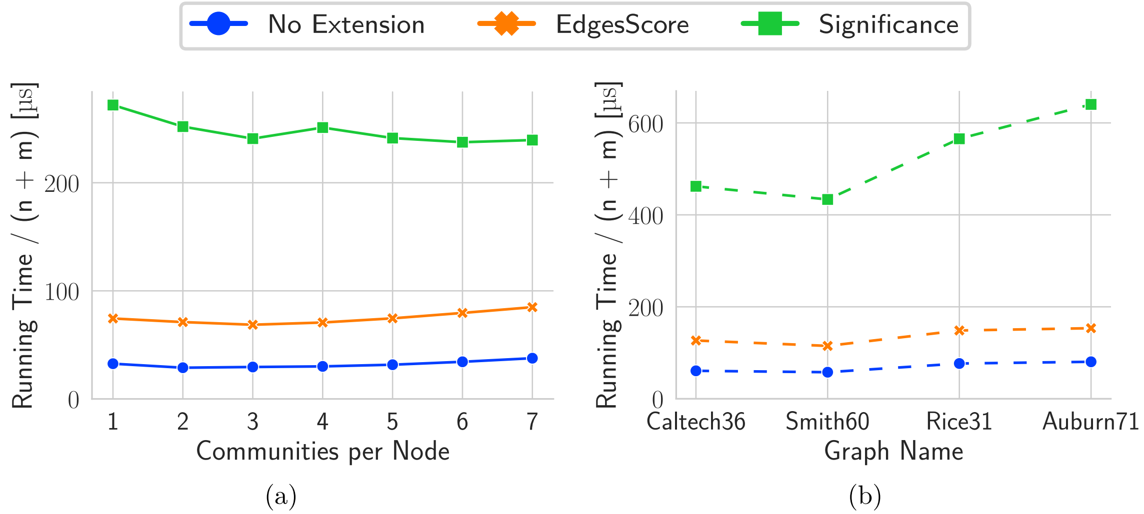
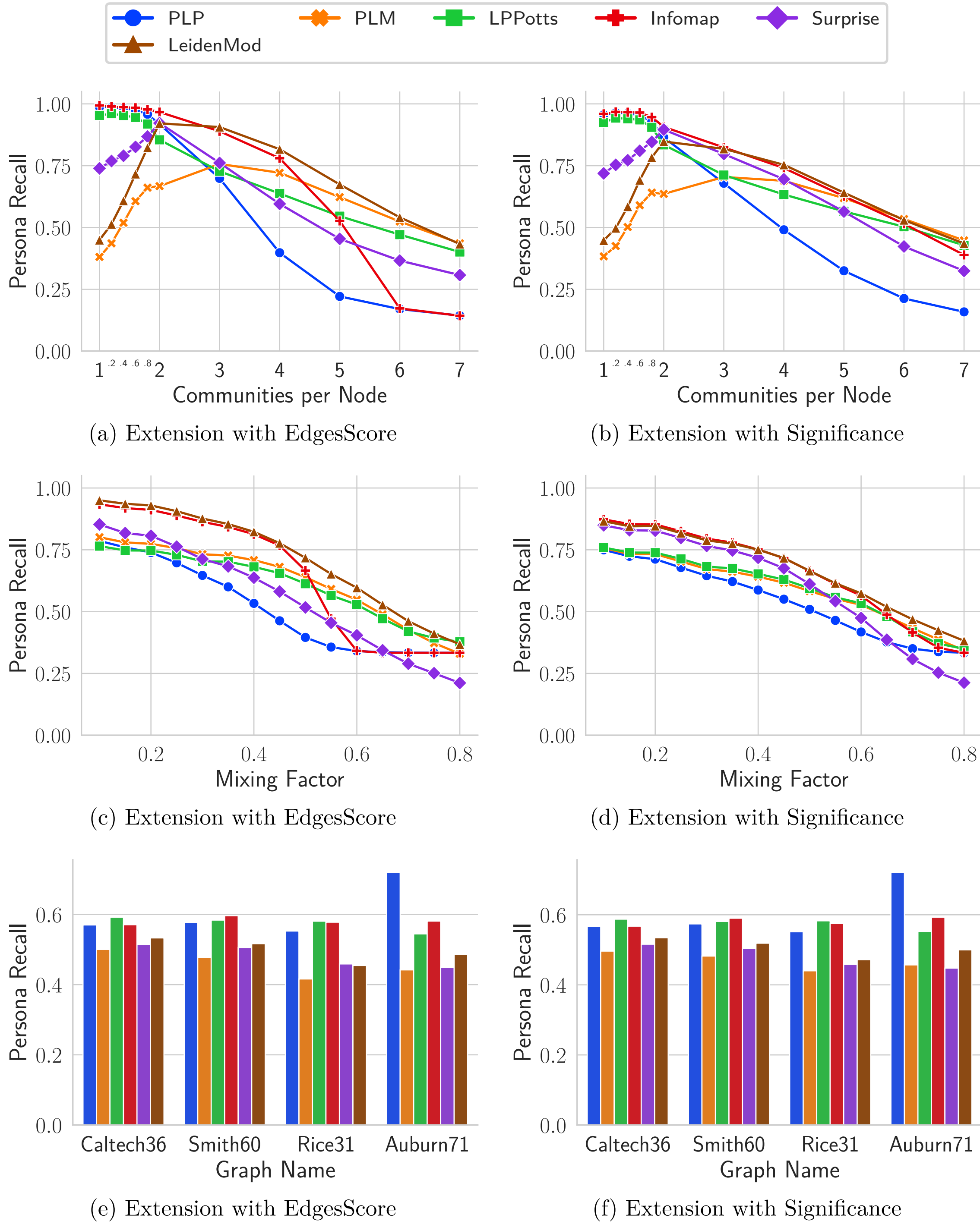


Figure 9.11: Running times of the extension process plus the running time of the local clustering algorithm, divided by the number of nodes and edges. The ego-net is extended using either EdgesScore or Significance. Figure (a) shows the results on the LFR graphs with a varying number of communities per node, Figure (b) shows the results on the Facebook graphs.

9.2 Local Clustering Algorithm

We compare various clustering algorithm to cluster the ego-net and analyze the detected clustering. We present results for both ego-net extension variants, EdgesScore and Significance. For each metric, we report the average value of all ego-nets. The only exception is the running time, where we present the (normalized) sum of the running times for all ego-nets.

Figure 9.12 depicts the persona recall for the LFR graphs and the Facebook graphs as described in Section 8.4.2. Figure 9.12a shows the persona recall for LFR graphs with varying numbers of communities per node, extending the ego-net using EdgesScore. The quality of all clustering algorithms decreases if the number of communities increases beyond three communities per node. For less than three communities per node, Infomap provides the best quality. For three or more communities per node, LeidenMod provides the best quality. There is no clustering algorithm that provides a high quality for all graphs. For one community per node, LeidenMod has a persona recall of only 0.449, the second worst of all algorithms. For two communities per node, the persona recall increases to 0.924. For more than four communities, the persona recall of Infomap decreases drastically. For six or more communities, Infomap is the worst algorithm, shared with PLP. PLM has a lower persona recall than LeidenMod for all graphs and PLP is worse than Infomap for all graphs. LPPotts and Surprise are never the worst algorithms, but also never the best. Both LeidenMod and PLM are based on modularity. The modularity is always maximized by diving the ego-net, even if all nodes form a single community. Because of that, the modularity based algorithms unnecessarily divide the ego-net for less than two communities per node, resulting in a low the persona recall.
Figure 9.12b shows the persona recall when the ego-net is extended using Significance. LPPotts, PLP and Surprise show the same progression as for EdgesScore. However, the persona recall with Significance is higher for the graphs with three or more communities per node. Compared to the extension with EdgesScore, Infomap has a noticeably higher persona recall for five or more communities per node, up to a factor of 3. LeidenMod performs worse for two to five communities per node compared to EdgesScore. For four communities per node however, LeidenMod with EdgesScore has a factor of 1.1 higher



(a) Extension with EdgesScore

(b) Extension with Significance

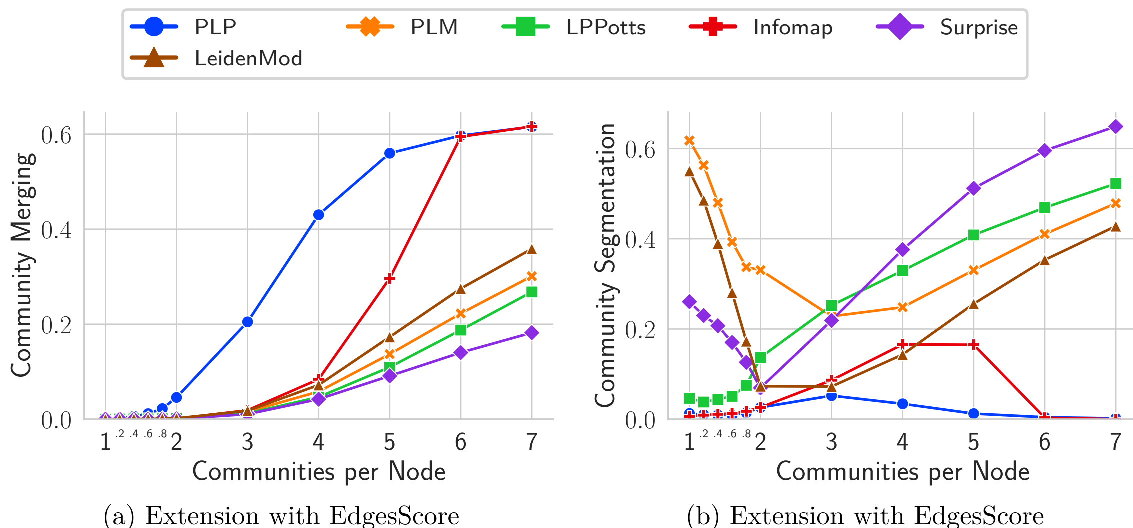
(c) Extension with EdgesScore

(d) Extension with Significance

(e) Extension with EdgesScore

(f) Extension with Significance

Figure 9.12: The persona recall of the ego-net clustering for different local clustering algorithms for LFR graphs and Facebook graphs. For the results on the left side, the ego-net was extended using EdgesScore, on the right side Significance was used instead.



(a) Extension with EdgesScore

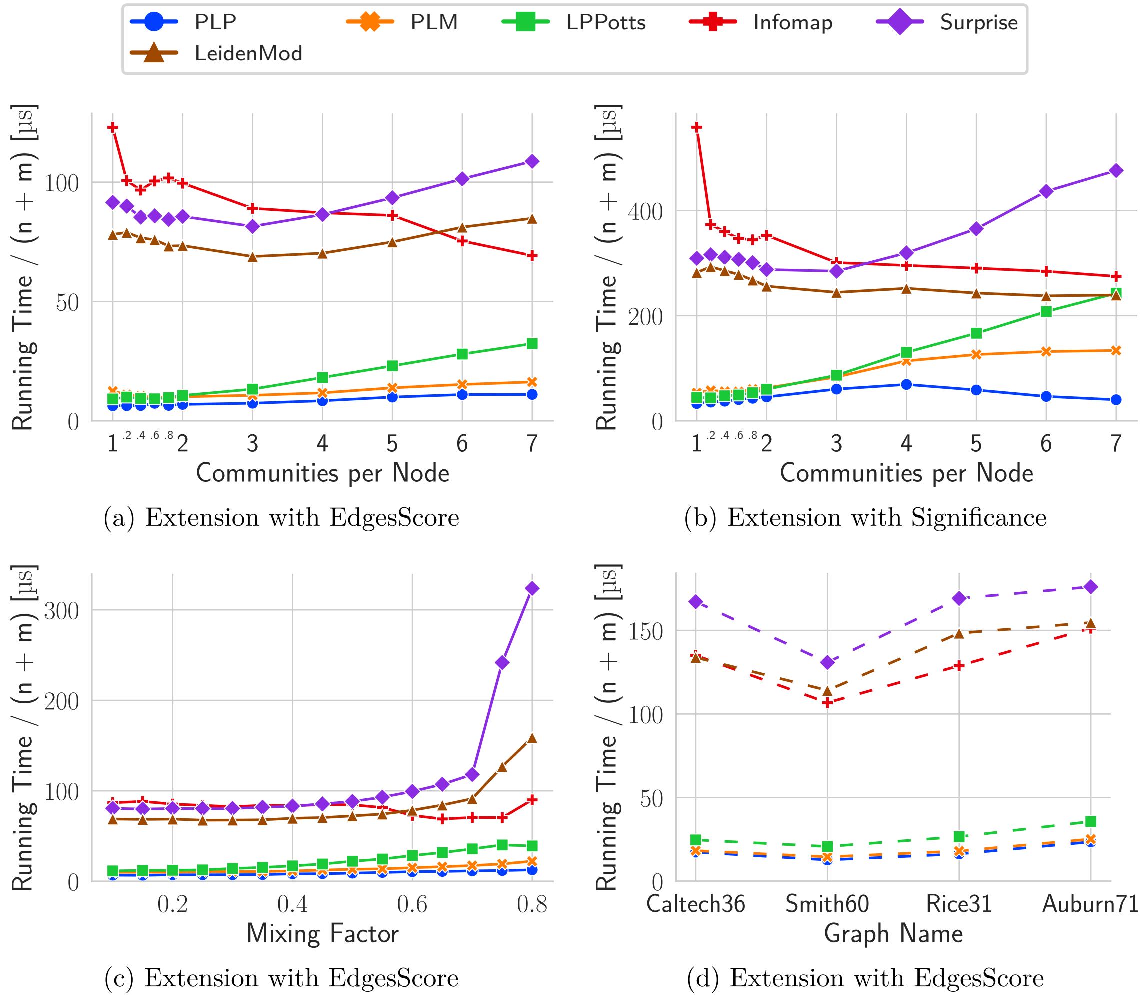
(b) Extension with EdgesScore

Figure 9.13: Clustering metrics of the result of the local clustering algorithms.

persona recall than LeidenMod with Significance. The best algorithms are Infomap for up to three communities per node and LeidenMod for more than three communities per node, the same as for EdgesScore. If we compare the best algorithm for each graph, EdgesScore has a higher persona recall than Significance for all graphs.

Figure 9.12c depicts the persona recall on the LFR graphs with varying mixing factor, using EdgesScore. As expected, the persona recall becomes worse for all algorithms if the mixing factor increases. Infomap becomes noticeably worse if the mixing factor exceeds 50%, dropping from the second best algorithm to the worst algorithm (shared with PLP) for a mixing factor of 60%. Infomap seems to have more problems than the other algorithms when it is applied to difficult graphs. For all graphs, except for a mixing factor of 80%, LeidenMod provides the highest quality. Figure 9.12d depicts the same graphs using Significance for the extension instead. In contrast to the extension with EdgesScore, the persona recall of Infomap does not decrease drastically for a mixing factor of more than 50%. Compared to EdgesScore, all other algorithms react similar to the increasing mixing factor. The persona recall of LeidenMod is always higher if we use EdgesScore instead of Significance.

Figure 9.12e and 9.12f show the persona recall on the Facebook graphs. There is no considerable differences between EdgesScore or Significance. This matches the results of the previous section that showed little difference between the two variants on the Facebook graphs. For all graphs, the three best clustering algorithms are PLP, LPPotts, and Infomap. The other three algorithms are noticeably worse, up to a factor of 0.83 for the graph Rice31. Comparing the results on the Facebook graphs and the LFR graphs, the ordering of the algorithms on the Facebook graphs matches the ordering on the LFR graphs with one to two communities per node. This is not a surprise, as the Facebook actually have less than two communities per node on average. Consequently, the modularity based algorithms PLM and LeidenMod most likely detect to many communities. The persona recall is noticeably lower on the Facebook graphs, never exceeding 0.6 except for PLP on one graph. In comparison, for the LFR graphs with communities with less than two communities, the algorithms PLP, LPPotts, and Infomap achieve a persona recall of more than 0.9. The lower persona recall indicates that we can not detect the ground-truth communities. We assume that the communities are not as clearly separated in the ego-net as it is the case for the LFR graphs. Additionally, the ground-truth communities are created from meta-data, so they do not necessarily match the structural communities.



(a) Extension with EdgesScore

(b) Extension with Significance

(c) Extension with EdgesScore

(d) Extension with EdgesScore

Figure 9.14: The normalized running time of the ego-net analysis for different local clustering algorithms.

We have seen which clustering algorithms provide a high persona recall on which graphs. In the following, we analyze in more detail the properties of the clusterings of the different algorithms. Figure 9.13 depicts the community merging score and the community segmentation score for LFR graphs with a varying number of communities per node. As described in Section 8.4.2, a high merging score indicates that multiple ground-truth communities are detected as one cluster. A high segmentation score indicates that each ground-truth community is detected as multiple clusters, i.e. its nodes are split between multiple clusters. For three or more communities per node, PLP has a high merging score, but a low segmentation score. This means that many ground-truth communities are merged together in a single cluster. Infomap has a similar progression, although the merging score only becomes large for five or more communities per node. On the other hand, Surprise has the lowest merging score, but the highest segmentation score for four or more communities per node. If there are many communities per node, Surprise divides each ground-truth community into multiple clusters. For less than two communities per node, the modularity based algorithms PLM and LeidenMod have a high segmentation score, so these two algorithms split the communities into multiple clusters. The reason is that the modularity is always maximized by splitting a graph into multiple clusters, so the communities are necessarily split if there are less than two communities per node. For 3 communities per node, the merging score for all algorithms except PLP is very low. At the same time, the segmentation score of Infomap and LeidenMod is also low,

while the segmentation score of PLM, LPPotts and Surprise is much higher in comparison. This confirms that Infomap and LeidenMod are the best clustering algorithms for three communities per node, as we have already seen by the persona recall. For LeidenMod and PLM, both scores increase linearly for an increasing number of communities per node. This looks like a “natural” progression, meaning the detected communities become worse as the difficulty of the detection increases. LeidenMod and PLM do not fall into an extreme as, for example, PLP, which merges many communities into a single cluster on the harder graphs. Because of this, LeidenMod and PLM seem like the most general-purpose algorithms, but only if there are at least two communities per node.

Figure 9.14 depicts the running times of the ego-net analysis, normalized by the number of nodes and edges. The ego-net analysis includes the creation of the ego-nets, the ego-net extension and the execution of the local clustering algorithm. Figure 9.14a gives the normalized running time on the LFR graphs with a varying number of communities per node, if we extend with EdgesScore. The clustering algorithms can be split into two sets: PLP, PLM, and LPPotts are relatively fast, while LeidenMod, Surprise, and Infomap are noticeably slower. PLP is the fastest algorithm for all graphs, followed by PLM and LPPotts. For five communities per node, PLM is slower than PLP by a factor of 1.4 and LPPotts is slower than PLP by a factor of 2.3. For five communities per node, LeidenMod is slower than PLM by a factor of 5.5. Figure 9.14b shows the normalized running time if we extend the ego-net with Significance. The ranking of the algorithms is mostly identical to EdgesScore. However, for seven communities per node, LPPotts is slower than Leiden. Because the extension with Significance uses the result of local clustering algorithm, the running time of the extension is also influenced by the clustering algorithm. For example, if the local clustering algorithm creates a clustering with many clusters, we check the significance of each candidate to each cluster (if the candidate has a neighbor in that cluster). If the clustering algorithm instead outputs only few clusters, we have to calculate the significance less often. The running time of Significance is higher than the running time of EdgesScore, which is expected since Significance extends and clusters three times. For three communities per node, the running time of LeidenMod with Significance is slower than EdgesScore by a factor of 3. For most graphs and algorithms, using Significance instead of EdgesScore increases the running time by a factor of 3 to 4. Figures 9.14c and 9.14d show the running time on LFR graphs with varying mixing factor and on the Facebook graphs. The ranking of the algorithms is similar as stated above, with PLP, PLM and LPPotts being noticeably faster than LeidenMod, Surprise and Infomap.

In conclusion, Infomap is the best clustering algorithm for small overlaps (two or fewer communities per node), but produces a low quality clustering for difficult graphs. LeidenMod is the best algorithm for a high overlap, but has problems dealing with less than two communities in the ego-net. Infomap, LeidenMod and Surprise are considerably slower than the other three algorithms, but only by a constant factor. No algorithm detects both slightly overlapping and highly overlapping graphs with a good quality. Comparing the variants of the ego-net extension, EdgesScore provides a higher quality than Significance for most graphs, especially for the LFR graphs with three to four communities per node. Extending with EdgesScore is also considerably faster. In the following sections, we use EdgesScore to extend the ego-net, and consider both Infomap and LeidenMod as the local clustering algorithm.

9.3 Connecting Personas

For each node, we connect its personas by inserting additional edges in the persona graph. We connect the personas so that the global clustering algorithm has a higher chance to repair mistakes made by the local clustering algorithm. Let P be the ego-persona graph of a node. We compare the following variants:

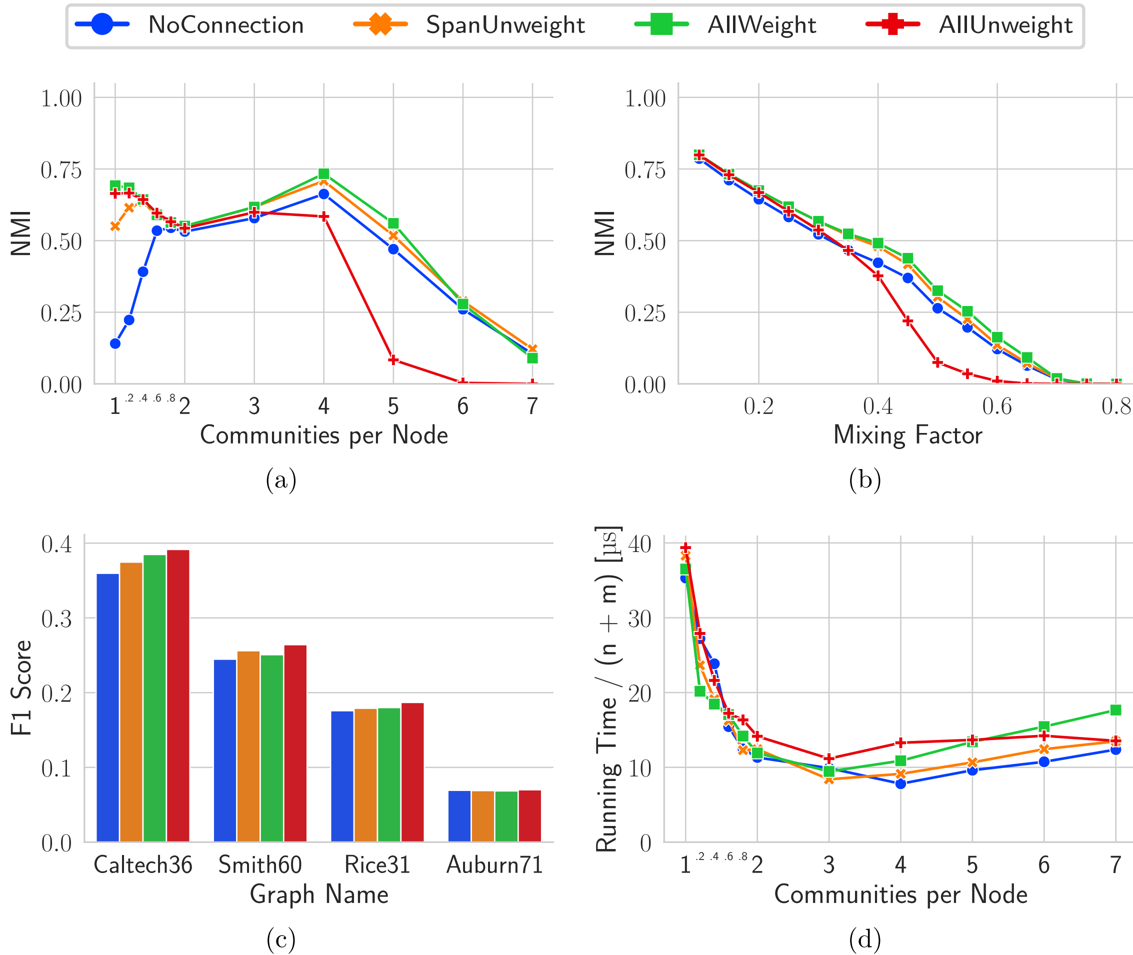


Figure 9.15: Connect personas of the same node using different strategies. The benchmarks use LeidenMod as the local clustering algorithm and Infomap as the global clustering algorithm. Shown are the NMI of the result cover for the LFR graphs with a varying number of communities per node (a), for the LFR graphs with varying mixing parameter (b) and the Facebook graphs (c). Figure (d) gives the normalized running time of the global clustering algorithm.

- **NoConnection:** No additional edges are inserted.
- **SpanUnweight:** Get a maximum spanning tree of P. For each edge of the spanning tree, insert an edge with weight 1.
- **AllUnweight:** For each edge in P, insert an edge with weight 1.
- **AllWeight:** For an edge $e = (p_1, p_2)$ with weight w in P, let $d(e) = \frac{w}{|p_1| \cdot |p_2|}$. For each edge e in P, insert an edge with weight $\frac{d(e)}{\max'_e d(e')}$.

We use LeidenMod as the local clustering algorithm, as it provides a high quality clustering for most graphs. In this section, we compare the detected cover against the ground-truth cover, therefore we have to choose a global clustering algorithm. We compare the choices for the global clustering algorithm in Section 9.4. Infomap provides high quality results for most graphs, so we us it as the global clustering algorithms for the experiments in this section.

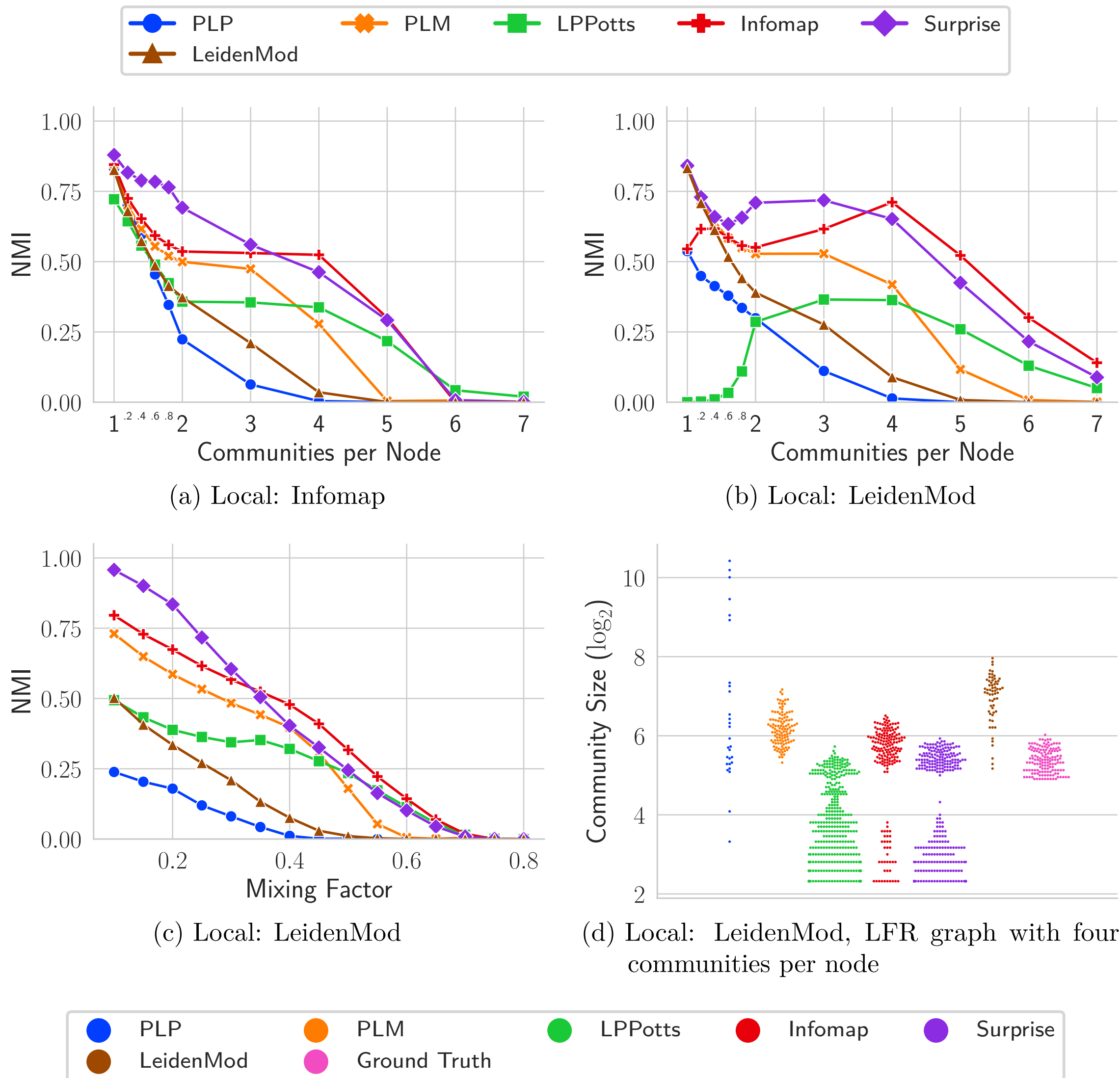
Figure 9.15 shows the results of the variants on the LFR and on the Facebook graphs. Figure 9.15a gives the NMI of the result cover. For one community per node, all connection

strategies improve the NMI drastically, by a factor of 4.4 to 5.4. As we use LeidenMod as the local clustering, too many personas are created for less than two communities per node, as we have seen in the previous section. As expected, connecting these personas improves the quality of the global clustering algorithm. For four or more communities, the variant AllWeight has a lower NMI than NoConnection, reaching an NMI of zero for six communities per node. We assume that for these graphs, there are often five or more personas per node. If these five personas are all connected in the ego-persona graph, then they form a clique that the global clustering algorithm detects as its own community. The strategy SpanUnweight improves the NMI for all graphs. The strategy AllWeight is the best strategy for two to five communities per node, but is worse than SpanUnweight for six communities per node. For seven communities per node, AllWeight is worse than NoConnection. For four communities per node, SpanUnweight increases the NMI compared to NoConnection by a factor of 1.06. For the same graph, AllWeight increases the NMI by a factor of 1.09.
Figure 9.15b gives the results on the LFR graphs with varying mixing factor. For all graphs, SpanUnweight is better than NoConnection, and AllWeight is better than SpanUnweight. AllUnweight is worse than NoConnection for a mixing factor of 0.4 or more.
Figure 9.15c shows the F1 Score on the Facebook graphs. For all graphs, all three connection strategies improve the F1 Score compared to NoConnection. Overall, AllUnweight provides the highest quality, followed by AllWeight and SpanUnweight.
Figure 9.15d depicts the running time of the global clustering algorithm, normalized by the number of nodes and edges of the graph. For some graphs with three or fewer communities per node, connecting the personas results in a lower running time compared to not connecting the personas on some graphs. This is not intuitive, as we have inserted additional edges into the persona graph. As we have seen, the quality of the detected communities is higher if we connect the personas. We assume that LeidenMod can detect these communities faster than the communities with lower quality. For four or more communities per node, SpanUnweight is slightly slower than NoConnection but faster than AllWeight. For six communities per node, SpanUnweight increase the running time by a factor of 1.1, AllUnweight by a factor of 1.26, and AllWeight by a factor of 1.41.

In conclusion, AllUnweight drastically decreases the quality on some graphs, so it is not a viable solution. Both AllWeight and SpanUnweight increase the quality in nearly all cases, AllWeight being slightly better for most graphs. However, AllWeight is not the best strategy for highly overlapping graphs, where it can even decrease the quality compared to NoConnection. AllWeight also has a higher running time than SpanUnweight. SpanUnweight consistently improves the quality and only has a low extra cost in the running time. Another advantage of SpanUnweight compared to AllWeight is that the global clustering algorithm does not have to support weighted graphs. We conclude that SpanUnweight is the most universal strategy and thus use it for all following benchmarks.

9.4 Global Clustering Algorithm

We compare the result covers for different global clustering algorithms, using either Infomap or LeidenMod as the local clustering algorithm. Figure 9.16a gives the NMI on the LFR graphs with a varying number of communities per node, using Infomap as the local clustering algorithm. For less than four communities per node, Surprise is the best global clustering algorithm, followed by Infomap. For two communities per node, Surprise has a NMI of 0.69, while Infomap has an NMI of 0.53. Both algorithms have the same quality for five or more communities per node. For six and seven communities per node, LPPotts has the highest NMI, although the cover has only a low quality. All other algorithms a strictly worse than Surprise on all graphs. Figure 9.16b depicts the NMI for the same graphs, but using LeidenMod as the local clustering algorithm. Again, Surprise provides the highest



(a) Local: Infomap

(b) Local: LeidenMod

(c) Local: LeidenMod

(d) Local: LeidenMod, LFR graph with four communities per node

Figure 9.16: Comparison of different global clustering algorithms. Figure (a) depicts the NMI using Infomap as the local clustering algorithm. For the figures (b)-(d), LeidenMod is the local clustering algorithm. Figure (d) shows the community sizes of the result cover.

quality for less than four communities per node and Infomap provides the highest quality for four or more communities per node. For five or more communities per node, the NMI of Infomap is between 0.05 and 0.1 higher than the NMI of Surprise. All other algorithms are strictly worse than Surprise for all graphs and strictly worse than Infomap on all graphs with two or more communities per node.

We see that Infomap and Surprise are the best global clustering algorithms. Now we compare the influence of the local clustering algorithm on the quality of the cover. Using Surprise as the global clustering algorithm, Infomap as the local clustering algorithm provides a higher quality than LeidenMod as the local clustering algorithm for the graphs with less than two communities per node. This is expected, as we showed in a previous section that Infomap produces a higher quality ego-net clustering for these graphs. For the LFR graph with an average of 1.6 communities per node, using Infomap locally results in an NMI of 0.792 while using LeidenMod locally results in an NMI of 0.651. For three or more communities per node, using LeidenMod locally instead of Infomap produces a higher quality for both Infomap and Surprise as the global algorithms. This confirms our

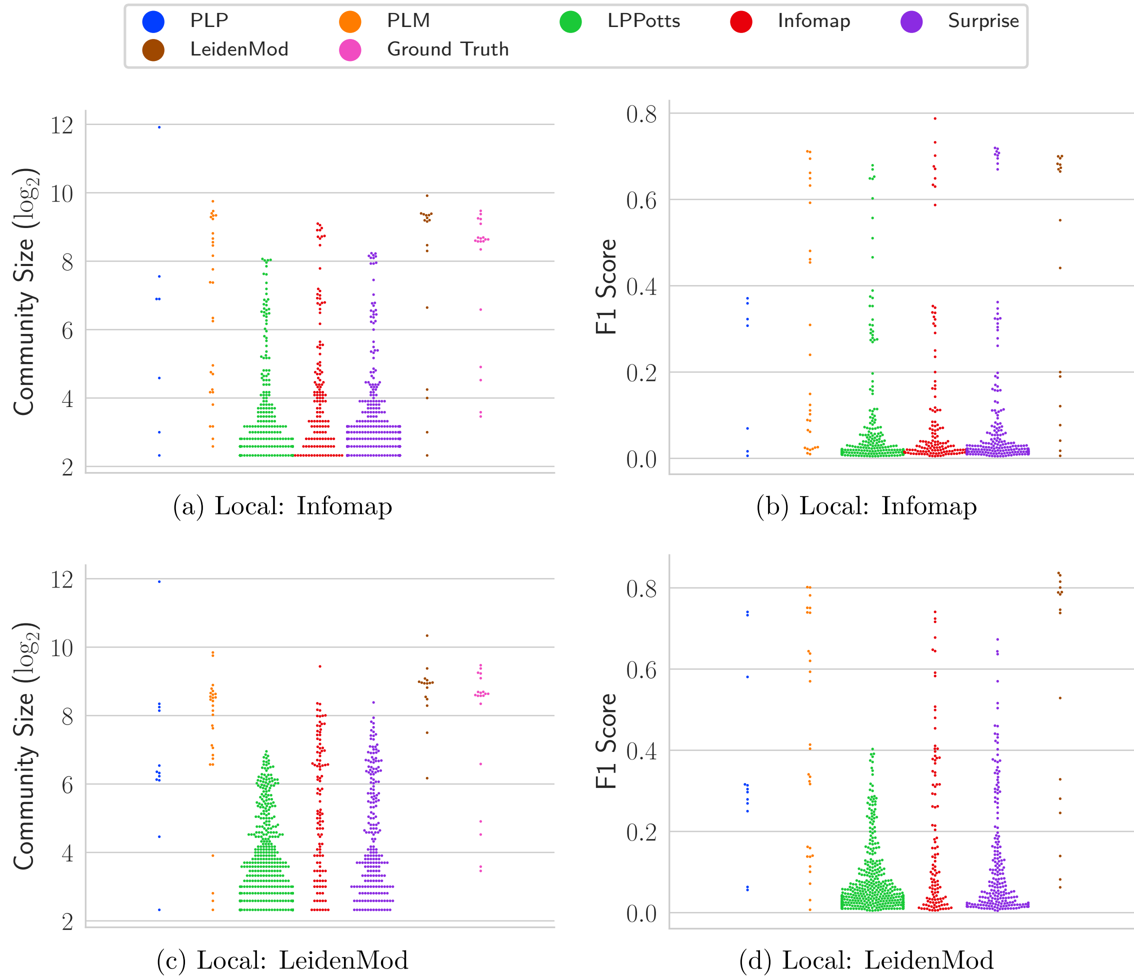


Figure 9.17: Results on the Facebook graph Rice31, using either Infomap (a + b) or LeidenMod (c + d) as the local clustering algorithm. Figure (a) and (c) give the sizes of the detected communities, while Figure (b) and (d) show the F1 Score of the detected communities.

conclusion that LeidenMod produces a higher quality ego-net clustering than Infomap for these harder graphs. For four communities per node and Infomap as the global algorithm, using LeidenMod locally results in an NMI of 0.72, while using Infomap locally results in an NMI of 0.516.
Figure 9.16c gives the NMI on the LFR graphs with varying mixing factor, using LeidenMod as the local clustering algorithm. Again, either Infomap or Surprise provide the highest quality for all graphs. For a mixing factor less than or equal to 30%, Surprise is the best global clustering algorithm. If the mixing factor is higher, Infomap provides a better quality, increasing the NMI by up to 0.07 compared to Surprise.
The NMI only provides a single value to measure the quality of a detected cover. To provide further insights about the clustering algorithms, we analyze the sizes of the detected communities as well as the number of detected communities. Figure 9.16d shows the sizes of the detected communities for the LFR graph with three communities per node and a mixing factor of 25%. This graph has a medium difficulty, where the local clustering algorithm provides a high quality, but is not able do detect the local clusters perfectly. Each point in the swarm plot corresponds to one community, the rightmost cluster showing the ground-truth cover. We present the cover of the first iteration of the benchmark, as depicting all ten iterations would make an evaluation difficult. We see that PLP detects

only a few very large communities. LPPotts detects way too many small communities and not enough large communities, indicating that LPPotts splits communities too easily. LeidenMod detects too few communities that are too large. The communities of PLM are a too large, up to double the size of the ground-truth communities. PLM also detects too few communities, indicating that multiple ground-truth communities are merged into one detected community. Both modularity based algorithms have the tendency to detect large communities. This may be caused by the resolution limit of modularity. Infomap and Surprise detect communities that match the ground-truth communities well in size and number. However, they also detect a number of small communities that have no counterpart in the ground-truth. Still, Infomap and Surprise detect communities that match the sizes of ground-truth communities the best, explaining their high NMI.
There is no combination of local and global clustering algorithm that provides the best quality for all LFR graphs. For the graphs with less than three communities per node, the combination Infomap/Surprise (local/global) detects the highest quality communities. For three communities per node, the algorithms LeidenMod/Surprise provide the best quality. For four or more communities per node, the highest quality is achieved by LeidenMod/Infomap. To keep the number of possible combinations manageable, we only consider Infomap/Surprise and LeidenMod/Infomap in the following chapters.

We now evaluate results on a Facebook graph. Because the ground-truth communities on the Facebook graphs do not perfectly represent the structural communities, it is difficult to compare the NMI for the algorithms on multiple graphs. Instead, we focus on one graph and analyze the size and F1 Score of the detected communities. Figure 9.17a shows the sizes of the detected communities on the Facebook graph Rice31, using Infomap as the local clustering algorithm. We see that there are very few ground-truth communities, most of them having a size between 2^8 and 2^{10}. As some algorithms detect many small communities that surely have a small F1 Score, the average F1 Score is very low. To analyze the quality of the detected communities, we look at the F1 Score of each community. Figure 9.17b gives the F1 Score for each community, again using Infomap as the local clustering algorithm on the graph Rice31. PLP detects very few communities and none with an F1 Score of 0.4 or more. PLM and LeidenMod detect few communities, roughly equal to the number of ground-truth communities. LPPotts, Infomap and Surprise detect many communities, especially small ones. As these small communities have no corresponding ground-truth communities, they have a very low F1 Score. All algorithms except PLP detect a few communities with a relatively high quality (F1 Score > 0.6). Figures 9.17c and 9.17d give the community sizes and F1 Scores if we use LeidenMod as the local clustering algorithm. The results are fairly similar to Infomap as the local clustering algorithm. The community sizes of LPPotts, Infomap and Surprise are more evenly distributed, i.e. there are more communities with a large size. LeidenMod provides the best quality, with most of the communities having a F1 Score around 0.8.
We conclude that the clustering algorithm combination LeidenMod/LeidenMod provides the highest quality on this graph. LPPotts, Infomap and Surprise detect many small communities, but the ground-truth contains nearly no such communities. However, it is possible that there are such small structural communities in the graph, but they are not found in the meta-data.

In conclusion, Infomap/Surprise and LeidenMod/Infomap seem to be the best algorithm combinations for the LFR graphs. The analysis on the Facebook graphs is difficult, as the number of detected communities differs greatly between the clustering algorithms. We focus our analysis in the following sections on the LFR graphs. Therefore, we only consider the two combinations Infomap/Surprise and LeidenMod/Infomap for the following experiments.

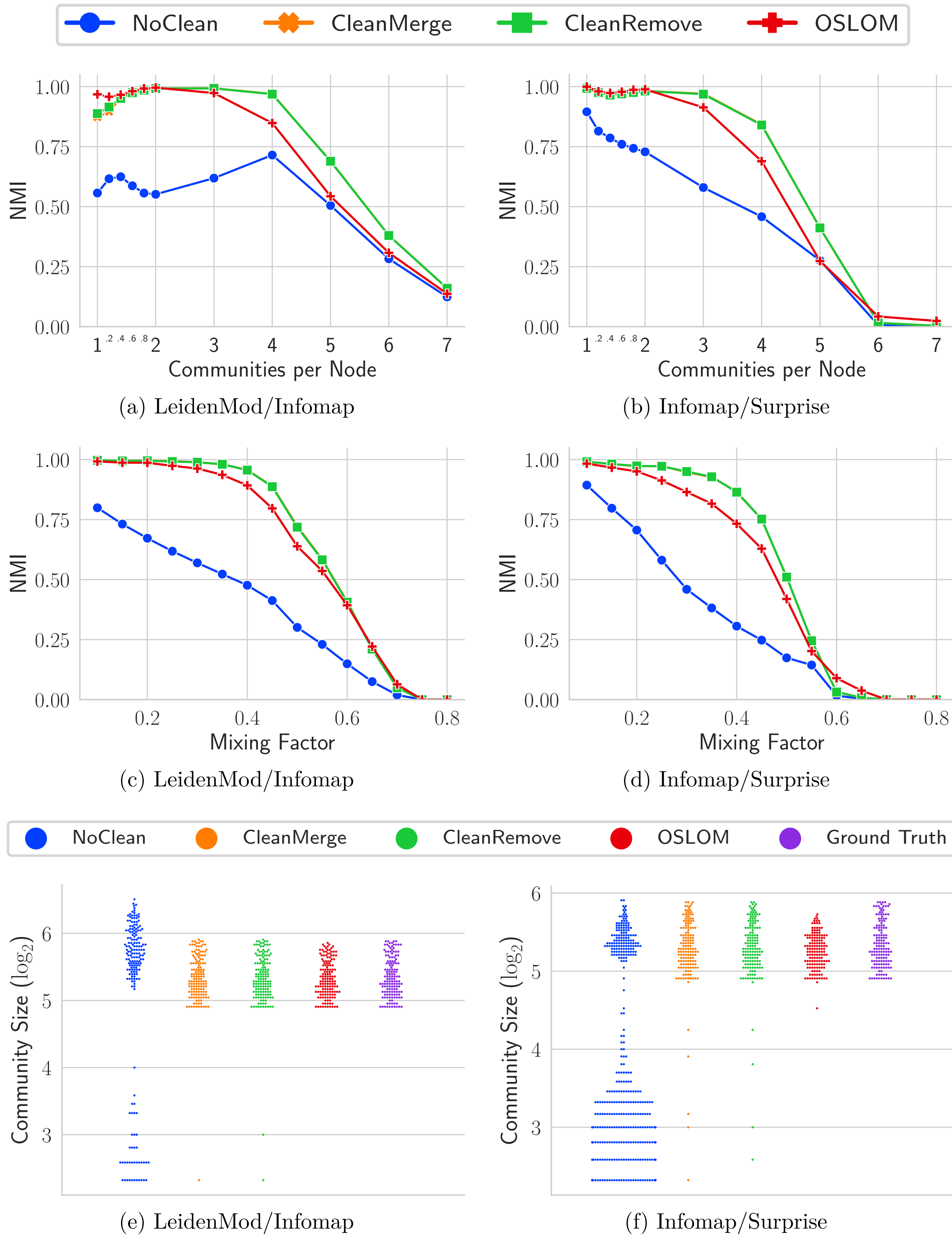
9.5 Community Clean-Up

After executing the ego-splitting framework, we clean up the detected cover using our proposed clean-up process based on statistical significance. We compare the following variants:

- **NoClean:** No clean-up step.
- **CleanRemove:** We clean up each detected community. Insignificant clusters are discarded.
- **CleanMerge:** We clean up each detected community. We try to merge insignificant communities to obtain significant communities.
- **OSLOM:** We run the OSLOM algorithm with the detected cover as the input.

Figure 9.18a depicts the NMI of the clean-up variants for the LFR graphs with a varying number of communities per node, using LeidenMod as the local and Infomap as the global clustering algorithm. For all graphs, CleanMerge and CleanRemove provide a nearly identical quality. Compared to the uncleaned cover, CleanMerge increases the NMI for all graphs. For two nodes per community, NoClean provides an NMI of 0.551 while CleanMerge provides an NMI of 0.989, an increase by a factor of 1.8. CleanMerge increases the NMI by a factor of 1.6 for four communities per node and by a factor of 1.3 for six communities per node. OSLOM also increases the NMI compared to NoClean for all graphs. OSLOM provides a higher quality than CleanMerge for the graphs with average number of communities less than or equal to two. For one community per node, OSLOM increases the NMI compared to CleanMerge by a factor of 1.09. For more than two communities per node, CleanMerge provides a higher quality than OSLOM. For six communities per node, CleanMerge increases the NMI compared to OSLOM by a factor of 1.18.
Figure 9.18b gives the NMI using Infomap as the local and Surprise as the global clustering algorithm. Again, CleanMerge and CleanRemove provide a nearly identical NMI for all graphs and improve the quality considerably compared to NoClean. For two or less communities per node, OSLOM has nearly the same quality as CleanMerge, with an NMI that is at best better than CleanMerge by a difference of 0.015. For two to five communities per node, CleanMerge provides a higher quality than OSLOM by a factor of up to 1.55. For six and seven communities, OSLOM has a higher NMI than CleanMerge, but both provide only a low quality result.
We see that the clean-up step considerably improves the quality of the detected communities, nearly perfectly recovering the ground-truth communities for some graphs. For the graphs with more than the two communities per node, our clean-up process provides a better quality than OSLOM.

Since the clean-up considerably increases the quality on some graphs, it is possible that the order of the local/global clustering algorithms changes. For example, algorithm combination A provides a lower quality than algorithm combination B without the clean-up, but the cleaned cover of A has a higher quality than the cleaned cover of B. To evaluate if this happens for the clustering algorithms LeidenMod/Infomap and Infomap/Surprise, we now compare the results of CleanMerge for the two algorithm combinations. For less than two communities per node, Infomap/Surprise provides a higher quality than LeidenMod/Infomap. This is expected because the quality of the uncleaned result of Infomap/Surprise is already considerably higher. For one community, LeidenMod/Infomap provides an NMI of 0.867, while Infomap/Surprise provides an NMI of 0.986. For more than two communities per node, the quality order turns around and LeidenMod/Infomap provides a higher quality, following the quality progression of their uncleaned covers. For four communities per node, LeidenMod/Infomap cleaned up with CleanMerge provides an NMI of 0.994, while Infomap/Surprise cleaned up with CleanMerge provides an NMI



(a) LeidenMod/Infomap

(b) Infomap/Surprise

(c) LeidenMod/Infomap

(d) Infomap/Surprise

(e) LeidenMod/Infomap

(f) Infomap/Surprise

Figure 9.18: The results of the clean-up step, comparing the uncleaned cover (blue) and the covers produced by the clean-up procedures. Shown are the NMI values (a - d) and the community sizes (e + f). For the plots on the left, Leiden-Mod/Infomap are the local/global clustering algorithms, for the plots on the right Infomap/Surprise are the clustering algorithms.

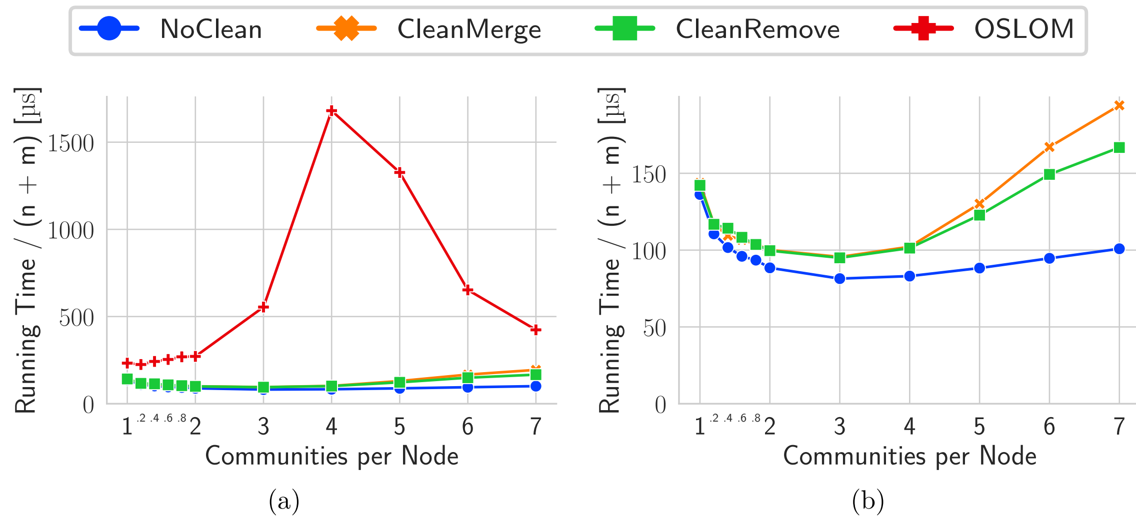
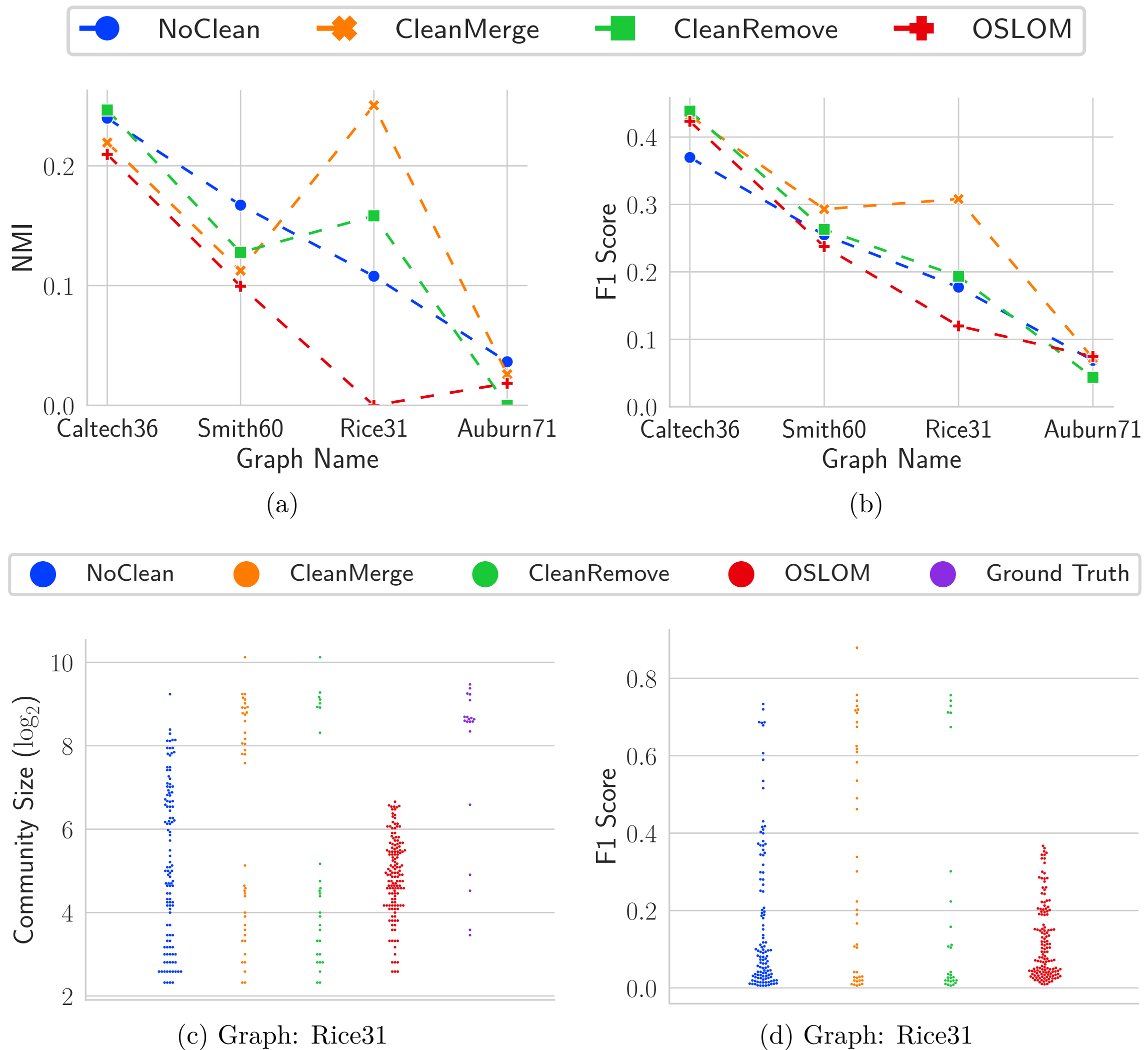


Figure 9.19: The normalized running times of the entire ego-splitting framework on the LFR graphs with a varying number of communities per node, comparing the clean-up procedures. The local clustering algorithm is LeidenMod and the global clustering algorithm is Infomap. Figure (b) shows the running time zoomed in on the fast versions.

of 0.847. In conclusion, the clean-up does not produce an unexpectedly high quality for one algorithm combination. Infomap/Surprise provides a higher quality for two or less communities per node and LeidenMod/Surprise provides a higher quality for three or more communities per node, for both the uncleaned and the cleaned covers.
Figures 9.18c and 9.18d show the NMI on the LFR graphs with varying mixing parameter, using either LeidenMod/Infomap or Infomap/Surprise as the local/global clustering algorithm combination. For both algorithm combinations, all clean-up variants improve the quality for all graphs. For a mixing factor of 50% and less, CleanMerge provides a higher quality than OSLOM. Only for a mixing factor of 60% and the algorithm combination Infomap/Surprise, OSLOM achieves a higher NMI. In this case however, even OSLOM does not provide a high quality result with only an NMI of 0.044.
Figures 9.18e and 9.18f depict the sizes of the detected communities on the LFR graph with three communities per node and mixing factor 25%. For both algorithm combinations, the uncleaned cover contains to many large communities but also contains some small communities that have no counterpart in the ground-truth. After the clean-up, the small communities have been discarded, and the detected communities match the ground-truth communities nearly perfectly.

Figure 9.19 depicts the normalized running time of the entire community detection, including the clean-up process. Using the OSLOM algorithm to clean up the cover increases the running time drastically, up to a factor of 18 for four communities per node. For four or less communities per node, CleanMerge is slower than NoClean by a factor of at most 1.25. For seven communities per node, CleanMerge is slower by a factor of 1.92. As expected, CleanMerge is slower than CleanRemove because it includes an additional step. For four or less communities per node, CleanMerge has nearly the same running time as CleanRemove, being slower by less than 1%. For seven communities per node, CleanMerge is slower than CleanRemove by a factor of 1.16. We see that OSLOM can drastically the running time, while our clean-up process CleanMerge increases the running time by a factor of less than 2 for all graphs.



(a) (b)

(c) Graph: Rice31 (d) Graph: Rice31

Figure 9.20: Results of the clean-up variants on the Facebook graphs. The local clustering algorithm is LeidenMod and the global clustering algorithm is Infomap. Shown are the NMI of the cover (a), the F1 Score of the cover (b), the sizes of the detected communities (c), and the F1 Scores of the detected communities (d).

The above results show that the clean-up process improves the quality on the synthetic graphs. Now we evaluate the effect of the clean-up process on the real-world graphs. We focus our analysis on the clustering algorithm combination LeidenMod/Infomap, as it provided the best quality on the synthetic graphs.

Figure 9.20 shows the results on the Facebook graphs, using the clustering algorithms LeidenMod/Infomap. For all graphs except Rice31, all three clean-up variants decrease the NMI of the cover. This shows again that the evaluation on the Facebook graph is difficult. We do not know why the clean-up step considerably improves the NMI on the graph Rice31, but decreases the NMI on the other graphs.

Figure 9.20b shows the F1 Score of the covers. CleanMerge increases the F1 Score on all graphs. CleanRemove provides a worse F1 Score than CleanMerge, showing that merging discarded communities increases the quality of the cover.

Figure 9.20c depicts the community sizes and Figure 9.20d depicts the F1 Scores of the detected communities, both for the graph Rice31. Using OSLOM decreases the average size of the communities, but keeps the number of communities relatively similar. The communities produced by OSLOM are much smaller than the ground-truth communities. Both CleanRemove and CleanMerge increase the number of large communities and decrease

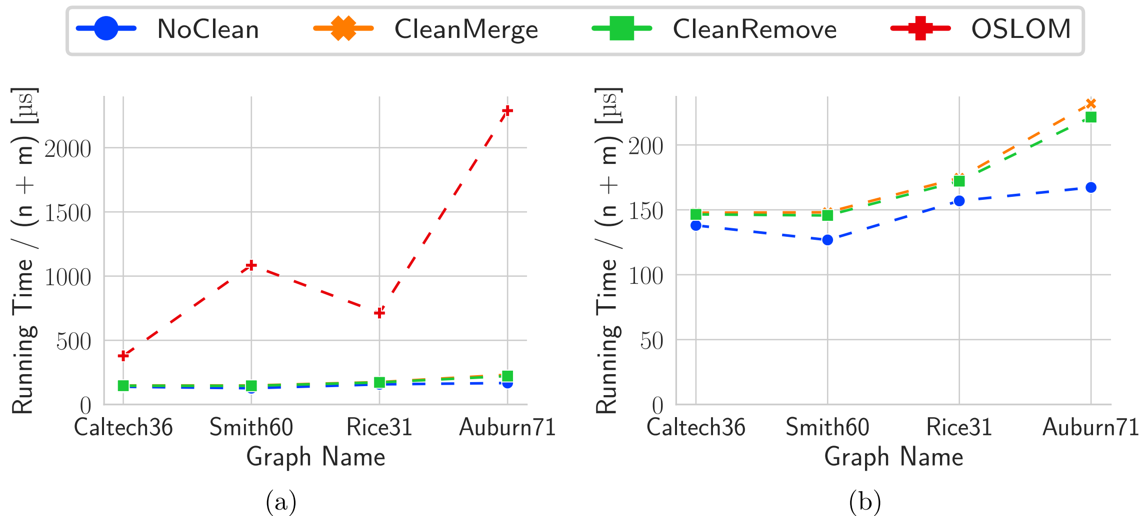


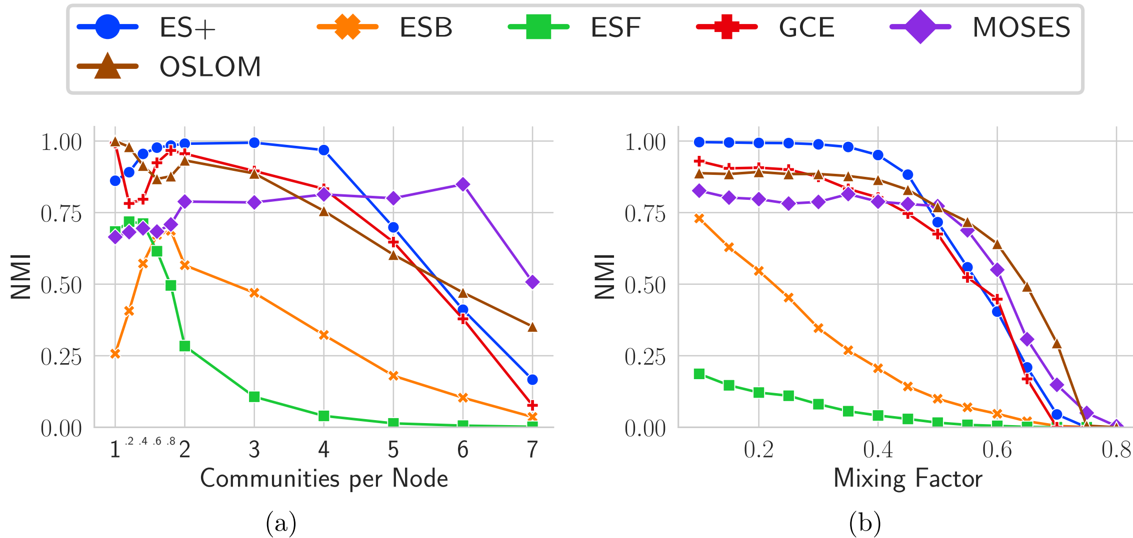
Figure 9.21: The normalized running times of the entire ego-splitting framework on the Facebook graphs, comparing the clean-up procedures. The local clustering algorithm is LeidenMod and the global clustering algorithm is Infomap. Figure (b) shows the running time zoomed in on the fast versions.

the number of small communities considerably, so that the sizes match the ground-truth much better. This results in much fewer communities with a low F1 Score. However, CleanMerge produces more large communities which have a similar size than the ground-truth communities, thus detecting more communities with a high F1 Score. This shows that we can create additional significant communities by merging discarded communities. Figure 9.21 gives the normalized running time on the Facebook graphs. As was the case for the LFR graphs, OSLOM is much slower than NoClean on some graphs, by a factor of 13 for the graph Auburn71. In contrast, CleanMerge is slower than NoClean by a factor of less than 1.5 for all graphs.

In conclusion, the clean-up procedure CleanMerge drastically improves the cover quality for the LFR graphs, nearly perfectly recovering the ground-truth on some graphs. On the real-world graphs, the clean-up step improves the quality only on some graphs. On the synthetic graphs, CleanRemove and CleanMerge provide nearly the same quality, but on the real-world graphs CleanMerge is clearly superior. CleanMerge provides a higher quality then the OSLOM algorithm for all graphs, while also being much faster than OSLOM. The additional running time of CleanMerge is moderate on all tested graphs, increasing the total running time by a factor of less than two for all graphs. We use the clean-up procedure CleanMerge for the following benchmarks.

9.6 Comparison with Other OCD Algorithms

We compare our optimized configuration of the ego-splitting framework with other overlapping community detection (OCD) algorithms and two other configurations of the ego-splitting framework. For clarity, we describe the optimized configuration in the following: We extend the ego-net using EdgesScore with the candidate score function $q(v) = \frac{{k_v^{in}}^2}{k_v}$. The maximum number of extended nodes is $o = 5 \cdot {n_e}^{0.5}$, the candidates are evaluated iteratively for $I_{max} = 10$ iterations, and the extension process is repeated $I_c = 3$ times. We use the local clustering algorithm LeidenMod to partition the ego-net. In the persona graph, we connect all personas of a node based on a maximum spanning tree in the ego-net. The global clustering algorithm is Infomap. After creating a cover from the clustering on



(a) (b)

Figure 9.22: NMI of the detected cover on the LFR graphs for different OCD algorithms.

the persona graph, we clean up that cover using our process CleanMerge.
We test three configurations of the ego-splitting framework in our experiments:

- **ES+** (Ego-Splitting Plus): Our optimized configuration as described above.
- **ESB** (Ego-Splitting Base): The ego-splitting framework without our proposed additional phases. The clustering algorithms are LeidenMod/Infomap, the combination that provided the best results in previous sections.
- **ESF** (Ego-Splitting Fast): The implementation of the ego-splitting framework given by Epasto et al. [ELPL17]. The local clustering algorithm is LPPotts with $\alpha = 0.1$, the global clustering algorithm is LPPotts with $\alpha = 0$.

We compare ego-splitting with other OCD algorithms that have shown to provide high quality results:

- **GCE**: Greedy clique expansion based on a simple community fitness function. We set $\alpha = 1.1$, because this value provided the best results in our benchmarks.
- **MOSES**: Seed expansion based on a stochastic block model.
- **OSLOM**: Local optimization to find statistically significant communities.

See Section 3.2 for a more detailed description of the algorithms.

Figure 9.22 shows the NMI of the cover detected by the different algorithms on the LFR graphs with a varying number of communities per node. ES+ provides the best quality for the graphs with 1.4 to 4 communities per node. For one community per node, OSLOM and GCE recover the ground-truth communities perfectly, while ES+ has an NMI of 0.879. For five or more communities per node, MOSES outperforms ES+ in terms of NMI. MOSES provides an NMI of 0.85 for six communities per node, while ES+ provides an NMI of 0.41. On the other hand, ES+ is better than MOSES for less than five communities per node, with an NMI higher by 0.2 for two communities per node. OSLOM has a higher NMI than ES+ for six and seven communities per node. GCE outperforms ES+ for one community per node by a factor of 1.14, but ES+ outperforms GCE for all other graphs. For four communities per node, GCE provides an NMI of 0.83, considerably lower than ES+ which provides an NMI of 0.97. ESF provides an NMI of around 0.7 for the graphs with 1.6 or less communities per nodes. For larger numbers of communities per node, the quality of ESF decreases drastically, providing an NMI of 0.28 for two communities per node and

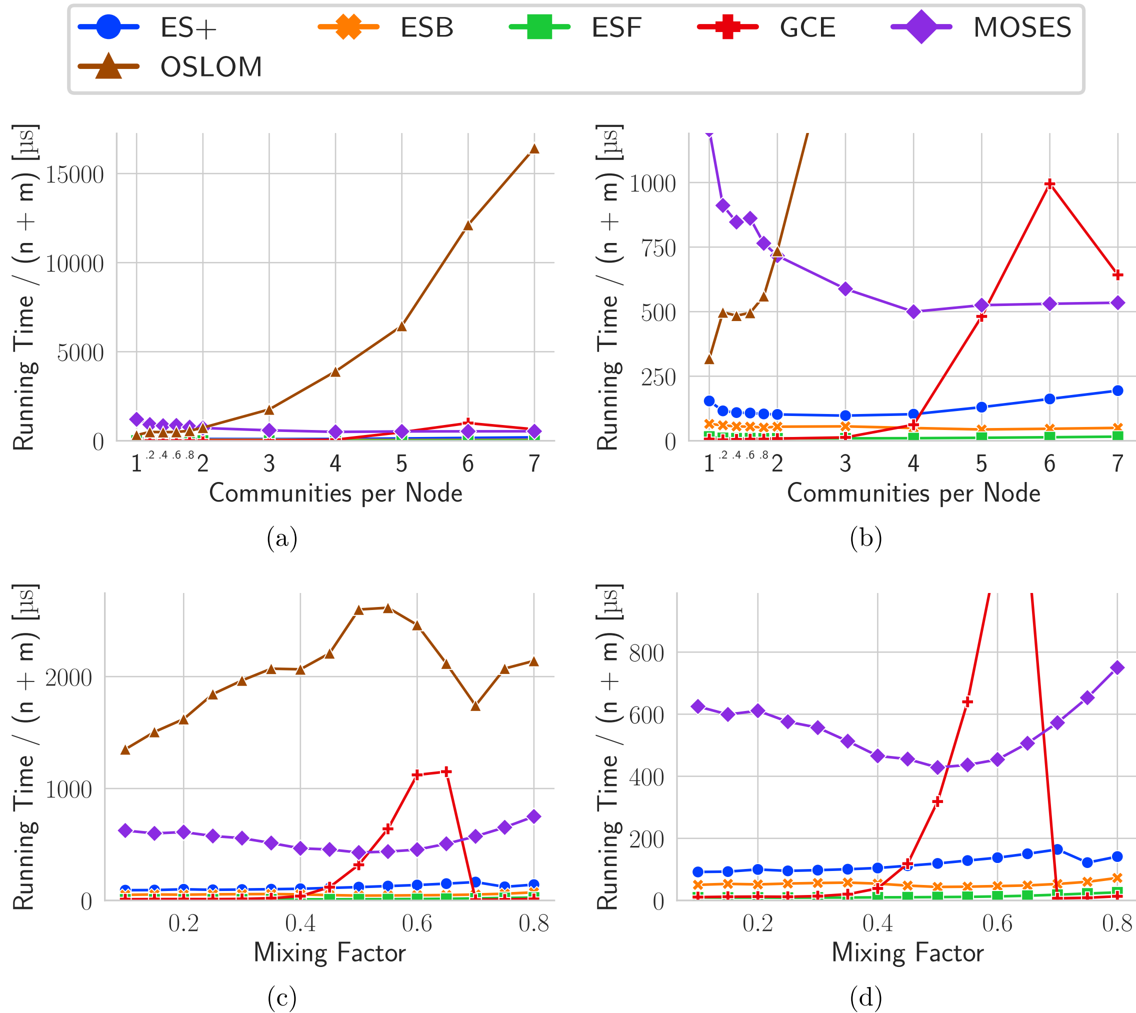


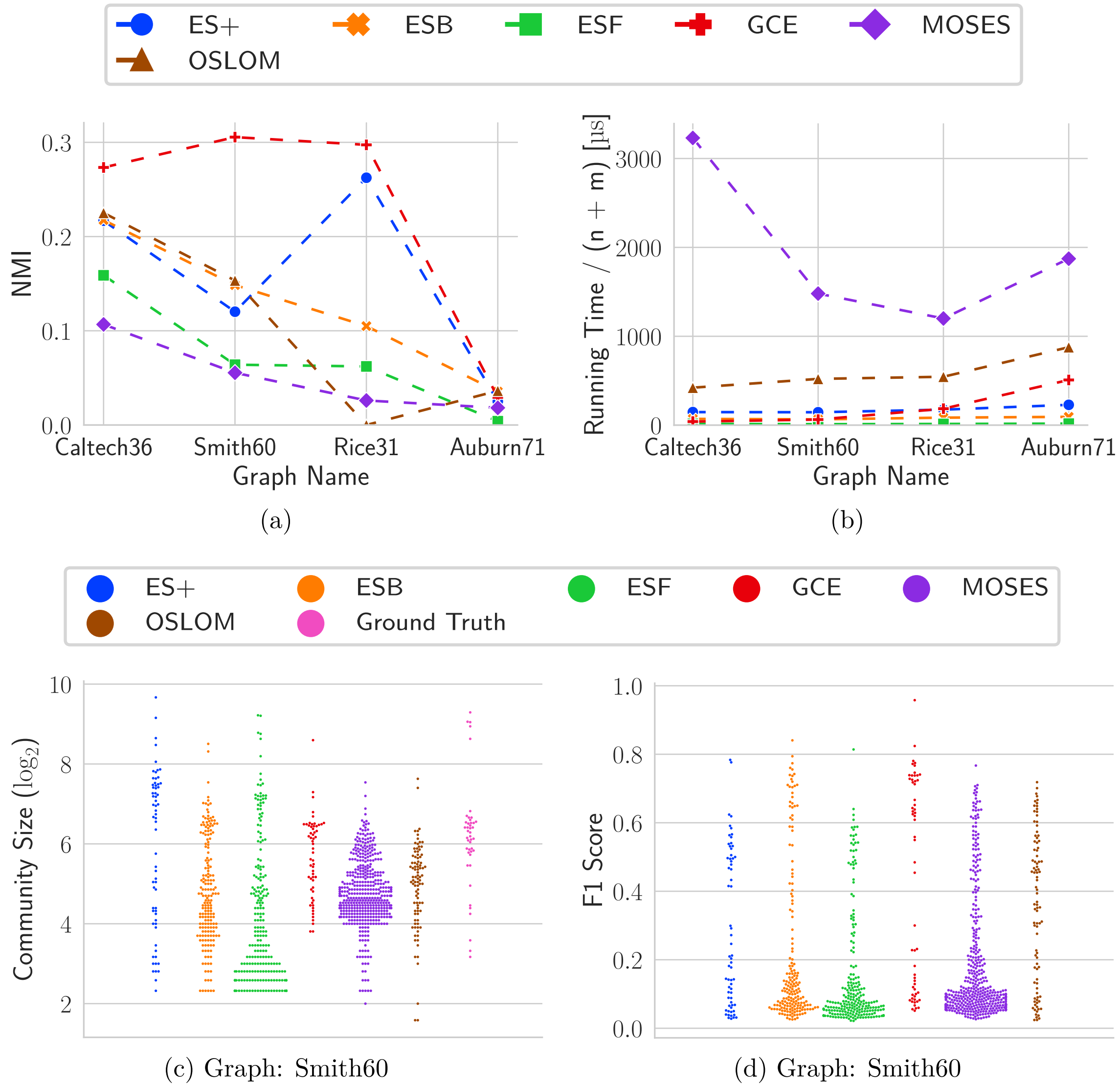
Figure 9.23: Running times of the community detection algorithms on the LFR graphs. Figure (b) and (d) are zoomed in graphs of (a) and (c).

providing an NMI of only 0.04 for four communities per node. ESB has an NMI of 0.26 for one community per node, drastically worse than all other algorithms. The reason is that the used local clustering algorithm (LeidenMod) does not provide a good quality for this graph. Without our additional phases, this low quality of the local evaluation can not be repaired, resulting in a low quality cover. ESB provides the best quality for the graph with an average of 1.8 communities per node, yielding an NMI of 0.69. For a higher number of communities per node, the quality declines drastically. For four communities per node, ESB has an NMI of 0.32. Comparing the variants of the ego-splitting framework for the highly overlapping graphs (more than two communities per node), we see that ESF performs extremely bad. ESB provides a higher quality, showing that the clustering algorithms LeidenMod/Infomap improve the quality considerably compared to LPPotts. However, ESB has still a much lower quality then ES+, with its NMI being up to 0.64 lower.

Figure 9.22b gives the NMI on the LFR graphs with varying mixing parameter. As expected, the quality of all algorithms decreases as the mixing factor increases. For a mixing factor of 40% or less, ES+ has the highest quality of all algorithms. For a mixing factor of 30%, the second best algorithm is OSLOM, which provides an NMI that is lower by 0.1 compared to the NMI of ES+. For a mixing factor of 50% and more, OSLOM provides the best quality, followed by MOSES, while ES+ and GCE have a similar quality. ESF has an extremely low quality, with an NMI of less than 0.2 for all graphs. ESB has a higher quality than

ESF, starting from an NMI of 0.73 for a mixing factor of 5%. The quality of ESB increases fast for an increasing mixing factor, providing an NMI of 0.21 for a mixing factor of 40%. On the same graph, ES+ has an NMI of 0.95.
Figure 9.23 gives the running times of the algorithms on the LFR graphs, normalized by the number of nodes and edges of the graph. The running time of OSLOM increases drastically if the number of communities per node increases. For seven communities per node, OSLOM is slower than ES+ by a factor of about 100 while it is only slower by a factor of 2 for one community per node. GCE is faster than ES+ for four or less communities per node by a factor of up to 30. For six communities per node however, GCE is slower than ES+ by a factor of 6. MOSES is slower than ES+ for all tested graphs. For one community per node, MOSES is slower than ES+ by a factor of 7.8 and for seven communities per node, it is slower by a factor of 2.7. On the graphs with varying mixing factor, OSLOM is slower than ES+ by a factor of at least 10 for all graphs. MOSES is slower than ES+ by a factor of at least 3.4 for all graphs. GCE is faster than ES+ for a mixing factor of 40% or less, by a factor of up to 9. For a mixing factor of 60%, the running time of GCE increases drastically and is slower than the running time of ES+ by a factor of 8.2. As expected, ESB is faster than ES+, and ESF is faster than ESB. Compared to ESF, ESB is slower by a factor of up to 6 and ES+ is slower by a factor of up to 11. ES+ is slower than ESB by a factor of up to 3. We see that using "good" clustering algorithms increases the running time considerably. The additional phases we introduced also increase the running time considerably. However, the running time still scales well for all graphs.
Our experiments show that for a medium number of communities per node, ES+ provides the best quality for all tested algorithms and nearly perfectly recovers the ground-truth communities. For less than two communities per node, ES+ can not recover the ground-truth perfectly. However, we have seen in previous sections that a different local clustering algorithm can improve the quality on these graphs. For hard graphs, i.e. a high number of communities or a high mixing factor, MOSES and OSLOM provide a higher quality than ES+. However, both algorithms are considerably slower than ES+ for all graphs, and provide a lower quality for the other graphs. GCE is faster than ES+ on the easier graphs, but has a very high running time on some of the harder graphs. ES+ also provides a higher quality than GCE for all but one graph. The other configurations of the ego-splitting framework are faster than ES+, but provide a much lower quality. Both the more sophisticated clustering algorithms and the additional phases improve the quality considerably. In conclusion, ES+ provides the best quality for many graphs. Additionally, ES+ has the best running time of the high quality algorithms, in the sense that the (normalized) running time is nearly constant for all LFR graphs. In contrast, MOSES is drastically slower for all graphs, and OSLOM and GCE require extremely high running times on some graphs.

We have shown that ES+ performs well on the synthetic graphs. Now we evaluate the OCD algorithms on the real-world graphs. Figure 9.24 shows the results on the Facebook graphs. Figure 9.24a gives the NMI of the detected covers. All algorithms provide a very low NMI on the graph Auburn71, less than 0.04. We assume that on this graph, the ground-truth communities do not actually represent the structural communities. For such low values, we can not compare the results in a meaningful way, so we ignore the graph Auburn71 in the following. GCE provides the highest NMI for all graphs. ES+ is the third best algorithm for the graphs Caltech36 and Smith60 and the second best algorithm for the graph Rice31. OSLOM provides the second best quality on the graphs Caltech36 and Smith60. On the graph Rice31 however, its NMI is only 0.02, meaning that the detected communities contain nearly no usable information. ESB has a slightly higher NMI than ES+ on the graphs Caltech36 and Smith60, but a considerably lower NMI on the graph Rice31. MOSES and ESF provide only a low quality on all graphs. The results show that the quality of the algorithms differs between the Facebook graphs, so there is no clear ordering of the



(c) Graph: Smith60

(d) Graph: Smith60

Figure 9.24: Results of the community detection algorithms on the Facebook graphs.

algorithms in terms of quality. To get more insights about the properties of the detected covers, we analyze the detected communities. Figure 9.24c depicts the sizes of the detected communities and Figure 9.24d depicts the F1 Score of the detected communities, both for the graph Smith60. Most of the ground-truth communities have a size of 50 to 120, but there are also five communities with a size of around 500. MOSES detects a large number of communities. Most of the detected communities have a low F1 Score. OSLOM detects mostly medium sized communities, which are smaller than most ground-truth communities. The F1 Score of the detected communities is relatively evenly distributed. GCE detects the lowest number of communities. Many of the communities detected by GCE match the sizes of the ground-truth communities well, and around halve of them have a high F1 Score. ESF detects many small communities that have no counterpart in the ground-truth. Consequently, the algorithm detects many communities with a very low F1 Score. Compared to ESF, ESB detects less small communities, resulting in less communities with a very low F1 Score. Also, ESB detects more high quality communities than ESF. ES+ detects only a few small communities, noticeably less than ESB, and also less communities overall. Consequentially, ES+ has less communities with a low F1 Score, compared to ESB. However, the best communities of ES+ have a lower quality than the best communities of ESB.

In conclusion, GCE seems to provide the highest quality result on the Facebook graphs. It is difficult to analyze the quality of the covers, as we can not guarantee that there are good ground-truth communities. We have seen that ESB and ES+ clearly improve the quality compared to ESF. The additional phases of the ego-splitting framework lead to the detection of less low quality communities.
Figure 9.24b gives the running time of the algorithms on the Facebook graphs. MOSES is the slowest algorithm, followed by OSLOM. For the Graph Auburn71, GCE is slower than ES+ by a factor of 3.2. Similar to the LFR graphs, ESF is the fastest ego-splitting algorithm, followed by ESB and lastly ES+. Compared to ESF, ESB is slower by a factor of up to 6, and ES+ is slower by a factor of up to 13.
As we have seen before, it is difficult to evaluate the quality of a cover on the Facebook graphs. Still, ES+ clearly provides a better quality than ESF and improves the quality compared to ESB at least for one graph. Similar to the results on the LFR graphs, ES+ is consistently faster than OSLOM and MOSES, and sometimes faster than GCE. The Facebook graphs are only slightly overlapping, with less than two communities per node on average. As we have seen on the synthetic graphs, ES+ provides high quality results even for four communities per node. We think that ES+ may outperform other algorithms on real-world networks that are highly overlapping.

In conclusion, ES+ provides a high quality on the synthetic graphs, clearly outperforming the other two ego-splitting algorithms. On many graphs, ES+ even has the highest quality of all tested algorithms. At the same time, ES+ has a stable running time that is not sensitive to the LFR graph parameters. On the Facebook graphs, we could not draw clear conclusions about the quality, but we have seen that the running time of ES+ is also stable and fast.

10. Conclusion

In this thesis, we have extended the ego-splitting framework by three steps: Ego-net extension, persona connection, and clean-up. The ego-net extension includes additional nodes in the ego-net, improving the structure of the communities. We connect the personas of a node to provide the global clustering algorithm with additional information about the relation of the nodes. The clean-up process uses statistical significance to remove weakly connected nodes from the detected communities, while including strongly connected nodes. Our experiments show that these additional phases increase the quality of the detected communities considerably. At the same time, we retain a relatively low running time.
We have evaluated the ego-net extension based on the structure of the ego-net. Our results show that the extension improves the structure of the communities, making them easier to detect. The extension process may be included in similar algorithms that analyze the ego-net, improving the quality of the detected local communities.
We have extensively tested different configurations of the framework, considering various clustering algorithms that have shown to provide high quality results. In our experiments, simple label propagation based algorithms could not compete with more sophisticated algorithms. For the ego-net analysis, the Infomap algorithm and the Leiden algorithm, using modularity as its fitness function, produced the highest quality results. For the community detection in the persona graph, the Infomap algorithm and the Leiden algorithm, using Surprise as a fitness function, discovered the best communities.
Our results show that we can improve the quality of communities considerably by adding a clean-up phase based on statistical significance. Unlike OSLOM, an algorithm based on the same measure, the clean-up process scales well even for complex graphs. Additionally, the clean-up process is not directly connected to the ego-splitting framework, so it can be used to clean up the detected communities of any overlapping community detection algorithm.

Compared to the previously proposed implementation of the ego-splitting framework, our implementation increases the running time, but we are able to provide a much higher quality. We have compared our implementation with state-of-the-art overlapping community detection algorithms. Our results show that our algorithm outperforms these algorithms in terms of quality for highly overlapping graphs. For two to four communities per node, we are able to nearly perfectly reconstruct the communities of synthetic benchmark graphs and outperform the state-of-the-art algorithms OSLOM and MOSES both in terms of quality and running time. Ego-splitting is faster than the other tested algorithms for most graphs. Moreover, the running time of ego-splitting is well predictable and does not depend on the structure of the graph. This is in contrast to the other algorithms, which react sensitively

to small changes of the parameters of the synthetic graphs.
We have shown that ego-splitting is able to detect high quality communities on highly overlapping graphs, while at the same time having a comparatively low running time. The ego-splitting framework remains flexible, allowing one to switch the clustering algorithms to meet quality or time constraints.

Further Work

The original ego-splitting framework is able to handle both directed and weighted graphs, given adequate clustering algorithms. The ego-net extension process is based on an unweighted and undirected graph. The process could be adjusted to also support weighted and directed graphs. The persona connection strategy is also not trivially adaptable to weighted and directed graphs, so further adjustments are needed.
For highly overlapping graphs, EdgesScore extends many external nodes, while Significance only extends the ego-net by very few nodes. Combining the two variants could make it possible to gain a high quality extension even for difficult graphs. To increase the number of nodes added by Significance, one could use ordered statistics to search for significant sets of nodes instead of evaluating each candidate in isolation, similar to the method used in OSLOM.
Our ego-net extension process can be used in any algorithm that analyzes the local communities of nodes. Further work could expand the evaluations of the extended ego-net, providing additional insight in the structure of network graphs, especially on the local level. The clean-up process is independent of the community detection algorithm. Applying the clean-up process to other algorithms, e.g. GCE, might improve the quality of the detected cover.
Synthetic benchmarks provide a useful tool to analyze the performance of community detection algorithms. However, the synthetic graphs, e.g. the graphs based on the LFR model, are still widely different than real-world network graphs. The implementation of the LFR graph generator could be extended to allow a heterogeneous distribution of the number of communities, as currently all nodes have the same number of communities. This would allow the creation of synthetic graphs that better resemble real-world graphs, bridging the gap between theoretical and practical results. Distributing the number of neighbors in a community according to the community size could also make the graphs more realistic.

Bibliography

[BGLL08] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[BKA+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 *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on*, pages 533–540. IEEE, 2014.

[Bur09] Ronald S Burt. *Structural holes: The social structure of competition.* Harvard university press, 2009.

[CNM04] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[CRGP14] Michele Coscia, Giulio Rossetti, Fosca Giannotti, and Dinor Pedreschi. Uncovering hierarchical and overlapping communities with a local-first approach. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 9(1):6, 2014.

[EB05] Martin Everett and Stephen P Borgatti. Ego network betweenness. *Social networks*, 27(1):31–38, 2005.

[ELM+15] Alessandro Epasto, Silvio Lattanzi, Vahab Mirrokni, Ismail Oner Sebe, Ahmed Taei, and Sunita Verma. Ego-net community mining applied to friend suggestion. *Proceedings of the VLDB Endowment*, 9(4):324–335, 2015.

[ELPL17] 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*, pages 145–154. ACM, 2017.

[FB07] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proceedings of the national academy of sciences*, 104(1):36–41, 2007.

[Fel81] Scott L Feld. The focused organization of social ties. *American journal of sociology*, 86(5):1015–1035, 1981.

[FLG+00] Gary William Flake, Steve Lawrence, C Lee Giles, et al. Efficient identification of web communities. In *KDD*, volume 2000, pages 150–160, 2000.

[Fre82] Linton C Freeman. Centered graphs and the structure of ego networks. *Mathematical Social Sciences*, 3(3):291–304, 1982.

[GN02] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[Gre07] Steve Gregory. An algorithm to find overlapping community structure in networks. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 91–102. Springer, 2007.

[Gre09] SteveB Gregory. Finding overlapping communities using disjoint community detection algorithms. In *Complex networks*, pages 47–61. Springer, 2009.

[HDF14] Darko Hric, Richard K Darst, and Santo Fortunato. Community detection in networks: Structural communities versus ground truth. *Physical Review E*, 90(6):062805, 2014.

[HHJ03] Petter Holme, Mikael Huss, and Hawoong Jeong. Subnetwork hierarchies of biochemical pathways. *Bioinformatics*, 19(4):532–538, 2003.

[HRW17] Michael Hamann, Eike Röhrs, and Dorothea Wagner. Local community detection based on small cliques. *Algorithms*, 10(3):90, 2017.

[HSWZ18] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. Distributed graph clustering using modularity and map equation. In *European Conference on Parallel Processing*, pages 688–702. Springer, 2018.

[LAH07] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5, 2007.

[LBA09] Pierre Latouche, Etienne Birmelé, and Christophe Ambroise. Overlapping stochastic block models. *arXiv preprint arXiv:0910.2098*, 2009.

[LC13] Conrad Lee and Pádraig Cunningham. Benchmarking community detection methods on social media data. *arXiv preprint arXiv:1302.0739*, 2013.

[LF09] Andrea Lancichinetti and Santo Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Physical Review E*, 80(1):016118, 2009.

[LFK09] Andrea Lancichinetti, Santo Fortunato, and Janos Kertesz. Detecting the overlapping and hierarchical community structure in complex networks. *New Journal of Physics*, 11(3):033015, 2009.

[LND16] Panagiotis Liakos, Alexandros Ntoulas, and Alex Delis. Scalable link community detection: A local dispersion-aware approach. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 716–725. IEEE, 2016.

[LRMH10] Conrad Lee, Fergal Reid, Aaron McDaid, and Neil Hurley. Detecting highly overlapping community structure by greedy clique expansion. *arXiv preprint arXiv:1002.1827*, 2010.

[LRR10] Andrea Lancichinetti, Filippo Radicchi, and José Jm Ramasco. Statistical significance of communities in networks. *Physical Review E*, 81(4):046110, 2010.

[LRRF11] Andrea Lancichinetti, Filippo Radicchi, José J Ramasco, and Santo Fortunato. Finding statistically significant communities in networks. *PloS one*, 6(4):e18961, 2011.

[LSS12] Min Chih Lin, Francisco J Soulignac, and Jayme L Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *Theoretical Computer Science*, 426:75–90, 2012.

[MGH11] Aaron F McDaid, Derek Greene, and Neil Hurley. Normalized mutual information to evaluate overlapping community finding algorithms. *arXiv preprint arXiv:1110.2515*, 2011.

[MH10] Aaron McDaid and Neil Hurley. Detecting highly overlapping communities with model-based overlapping seed expansion. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 112–119. IEEE, 2010.

[MMG+07] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM, 2007.

[MR95] Michael Molloy and Bruce Reed. A critical point for random graphs with a given degree sequence. *Random structures & algorithms*, 6(2-3):161–180, 1995.

[New06] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[OB14] Mark Ortmann and Ulrik Brandes. Triangle listing algorithms: Back from the diversion. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 1–8. Society for Industrial and Applied Mathematics, 2014.

[PDFV05] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *nature*, 435(7043):814, 2005.

[Pei15] Tiago P Peixoto. Model selection and hypothesis testing for large-scale network models with overlapping groups. *Physical Review X*, 5(1):011033, 2015.

[RAB09] Martin Rosvall, Daniel Axelsson, and Carl T Bergstrom. The map equation. *The European Physical Journal Special Topics*, 178(1):13–23, 2009.

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[RB06] Jörg Reichardt and Stefan Bornholdt. Statistical mechanics of community detection. *Physical Review E*, 74(1):016110, 2006.

[RB08] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[RG10] Bradley S Rees and Keith B Gallagher. Overlapping community detection by collective friendship group inference. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 375–379. IEEE, 2010.

[RMH11] Fergal Reid, Aaron McDaid, and Neil Hurley. Partitioning breaks communities. In *2011 International Conference on Advances in Social Networks Analysis and Mining*, pages 102–109. IEEE, 2011.

[RN10] Peter Ronhovde and Zohar Nussinov. Local resolution-limit-free potts model for community detection. *Physical Review E*, 81(4):046114, 2010.

[SH15] Sucheta Soundarajan and John E Hopcroft. Use of local group information to identify communities in networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 9(3):21, 2015.

[SSM14] Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. *arXiv preprint arXiv:1403.3005*, 2014.

[TAD15] Vincent A Traag, Rodrigo Aldecoa, and J-C Delvenne. Detecting communities using asymptotical surprise. *Physical Review E*, 92(2):022816, 2015.

[TMP12] Amanda L Traud, Peter J Mucha, and Mason A Porter. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, 391(16):4165–4180, 2012.

[TVDN11] Vincent A Traag, Paul Van Dooren, and Yurii Nesterov. Narrow scope for resolution-limit-free community detection. *Physical Review E*, 84(1):016114, 2011.

[TWvE18] Vincent Traag, Ludo Waltman, and Nees Jan van Eck. From Louvain to Leiden: guaranteeing well-connected communities. *arXiv preprint arXiv:1810.08473*, 2018.

[WF+94] Stanley Wasserman, Katherine Faust, et al. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[XKS13] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *Acm computing surveys (csur)*, 45(4):43, 2013.