# Learned Agglomerative Graph Clustering

September 20, 2022

Daniel Hammer

4075624

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
M. Sc. Marcelo Fonseca Faraj

# Acknowledgments

I want to thank Christian Schulz and Marcelo Fonseca Faraj for their guidance and continuous stream of great feedback and new inspirations. It was a great experience working with both of you on this project.
I am grateful for all the opportunities my parents give me and the support of my family and friends.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidelberg, September 20, 2022

Daniel Hammer

# Abstract

The task of graph clustering is to partition the graph into disjoint clusters of nodes with dense intra-cluster connections and sparse inter-cluster connections. These natural groups often reveal structural or intrinsic information about the data represented by the graph. There are different metrics to evaluate the quality of a clustering, such as conductance and modularity. Many algorithms tackle the problem of graph clustering, some based on spectral theory, others in combinatorial approaches, others in machine learning techniques, and so forth. An especially successful algorithm for graph clustering is greedy agglomeration, in which edges of the graph are greedily selected and contracted based on the modularity metric. In this work, we make a first attempt at combining machine learning techniques with the greedy agglomeration framework to build a clustering algorithm that is independent of a specific metric. We propose a supervised learning-based technique for finding clusters inside of graphs by utilizing structural edge properties only. We achieve in 65% of our test instances a higher score in the performance metric than greedy agglomeration and in around 20% higher modularity.

# Abstract (German)

Graph Clustering ist das Partitionieren eines Graphen in disjunkte Cluster, die viele Verbindungen innerhalb und wenige Verbindungen nach außen aufweisen. Diese natürlichen Gruppen können oft strukturelle oder versteckte Informationen der in einem Graphen dargestellten Daten enthüllen. Für die Messung der Qualität eines Clusterings gibt es verschiedene Metriken wie beispielsweise Conductance und Modularity. Es existieren verschiedenste Algorithmen um ein Clustering zu bestimmen, manche basieren auf Spektralanalysen, andere wiederrum auf kombinatorischen Ansätzen oder nutzen Machine Learning Techniken. Ein besonders erfolgreicher Algorithmus ist greedy agglomeration. Hier werden Kanten eines Graphs mit einer Greedy Policy basierend auf der Modularity Metrik gewählt und kontrahiert. In dieser Arbeit versuchen wir erstmals Machine Learning Techniken mit dem greedy agglomeration Framework zu kombinieren. Hierbei ist das Ziel einen Algorithmus zu schaffen, der unabhängig von einer spezifischen Metrik arbeitet. Der vorgestellte Algorithmus nutzt ausschließlich strukturelle Eigenschaften von Kanten um, mithilfe von Supervised Learning, Cluster in einem Graphen zu finden. In 65% der Testinstanzen erreichen wir einen höheren Wert in der Performance Metrik als greedy agglomeration. In etwa 20% eine höhere Modularity.

# Contents

CHAPTER 1

# Introduction

A graph clustering is a partition of a graph into groups or clusters that are very densely connected inside and very sparsely connected to other clusters. These simple requirements for a clustering allow a natural grouping of nodes in a graph. Finding the natural groups by clustering is in many cases a great aid for extracting insights and knowledge out of structured information in the form of a graph [1, 2]. These groups can help to identify groups of interest in social networks (e.g. terrorism groups), automatic malware detection [3], or discovering the structure of the internet [4]. In bioinformatics, clustering can help in analyzing gene expression data [5] or identifying repetitive sequences in DNA [6]. As any data with a relational or structural nature can be represented by a graph, clustering can be used in many different settings. As the problem has relevance there is a requirement for measuring the quality of a graph clustering to compare it against other possible solutions. For this purpose, multiple metrics exist that have different ways of qualifying properties like inter-cluster edges running between clusters or cluster sizes. However, most of the metrics are NP-hard to optimize [7]. With this, there is a need for different approximation algorithms [8, 9]. Spectral methods in which eigenvalues related to the graph are utilized for separating the graph into clusters have been used for a long time. For huge graphs, these approaches often suffer from high computational costs of finding eigenvalues and vectors. Algorithms that use combinatorial ideas are not dependent on eigenvalue decomposition, but instead, follow a set of predefined decision rules. This allows the algorithms to calculate a clustering in less time and thus makes even bigger graphs feasible for graph clustering. Especially muli-level Algorithms can cluster very huge graphs in a short amount of time [10]. With the advance in machine learning, learning-based techniques are also being used for graph clustering [11]. An approach that optimizes for the metric modularity directly is greedy agglomeration. This hierarchical algorithm performs contractions of nodes connected by edges by selecting the contraction, which yields the biggest improvement in modularity.

## 1.1 Our Contribution

We propose an algorithm combining supervised learning with the general algorithmic structure of greedy agglomeration. The learning technique extracts a decision policy for clustering from training data produced from graph clusterings. This policy only relies on structural information of edges like local node degrees or the size of the graph. With this technique, we can produce non-trivial clusterings and in some instances achieve better results than the original greedy agglomeration. In 65% of our test instances, we achieve higher scores in the performance metric. In 20% of the instances, our algorithm has higher modularity than the greedy agglomeration algorithm that aims to maximize modularity.

## 1.2 Structure

The remainder of this thesis is organized as follows. In Chapter 2 we specify definitions and knowledge required to follow the thesis and understand the approach. In Chapter 3 we discuss different works that address the problem of graph clustering and the different natures of methods used. In Chapter 4 we present our proposed algorithm. Additionally, to the clustering process of a graph, we introduce an important augmentation technique to prepare clustering data appropriately for training. This Chapter is also a detailed explanation of all features we use in any experiment and the complexity of all steps. In Chapter 5 we showcase how different factors like model choice, selected features, or hyperparameters affect our results and try to draw the first conclusions. In Chapter 6 we summarize the insights of the experiments. As a final remark, we point out some steps for improving our algorithm and further ideas.

CHAPTER 2

# Fundamentals

In the following, we define relevant terms, the problem of graph clustering, and introduce two machine learning concepts.

## 2.1 General Definitions

### 2.1.1 Graph

We denote a *graph* as $G = (V, E)$ consisting of a set of *nodes* (or vertices) V and a set of *edges* E connecting nodes in V with $n = |V|$ and $m = |E|$. We only consider graphs with *undirected* edges between nodes. Let $\omega_V : V \to \mathbb{R}_{\leq 0}$ be a *node weight* function and $\omega_E : E \to \mathbb{R}_{\leq 0}$ be an *edge weight* function. Let $iew : V \to \mathbb{R}_{\leq 0}$ be the *internal edge weight* of a node. A *subgraph* $S = (V', E')$ of graph $G$ has nodes $V' \subseteq V$ and edges $E' \subseteq (V' \times V')$. $S$ is an *induced* subgraph if $E' = E \cap (V' \times V')$. The *neighborhood* $N(v)$ of a node $v \in V$ is defined as $N(v) := \{u \in V : \{v, u\} \in E\}$. The *degree* of a node $v \in V$ is the number of neighbors this node has $deg(v) := |N(v)|$. We define the maximum degree in a graph as $\Delta := \max_{v \in V} deg(v)$. A matching $M \subseteq E$ in the graph G is a set of *matched* edges that have pairwise no common node. Every node that is adjacent to a matched edge is also called *matched*. The matching $M$ is maximal if no edge of the graph can be inserted in $M$ without matching a node a second time. A *cut* $(C, V \setminus C)$ with $C \subseteq V$ is a partition of a graph in two blocks. The *edge-cut* $m(C, V \setminus C)$ is the sum of the weights of all edges crossing the cut (i.e. $e = \{u, v\}$ with $u \in C$ and $v \notin C$). The *contraction* of two nodes $u, v \in V$ adds a new node $v'$ to $V$ and removes $u$ and $v$ from the node-set. The new node has the node weight $\omega_V(v') = \omega_V(u) + \omega_V(v)$. If an edge $e \in E$ connects the contracted nodes with a third node $w \in V$, $e$ is replaced by the edge $e' = \{v', w\}$. The edge weight of $e'$ is either the same as the removed edge $e$, if only one of the nodes is connected to $w$, $\omega_E(e') = \omega_E(e)$ or if the second node also has an edge $\widetilde{e}$ to $w$ the sum of the edge weights $\omega_E(e') = \omega_E(e) + \omega_E(\widetilde{e})$. If the edge $\hat{e} = \{u, v\}$ exists before

the contraction, it is now contained inside of the new node $v'$ and is removed from the edge set $E$. It is now an *internal edge* and $iew(v') = iew(u) + iew(v) + \omega_E(e)$ the internal edge weight of $v'$.

## 2.1.2 Graph Clustering

A graph clustering $C = \{C_1, \ldots, C_k\}$ is a division of the nodes of a graph into a not initially specified number of $k$ disjoint clusters $C_i$ which are sparsely connected and densely connected internally. A clustering is considered trivial if only one cluster exists containing all nodes (and thus all edges) or if every node is its own cluster (singletons). To measure the quality of a clustering there exist multiple metrics which we discuss next.

### Clustering Metrics

For readability we use the following notation for a given graph clustering $\mathcal{C} = \{C_1, \ldots, C_k\}$. Let $m(\mathcal{C})$ be the number of edges contained inside of all clusters (intra-cluster) and $m(\overline{\mathcal{C}})$ the number of node pairs not adjacent inside of clusters. Let $\overline{m}(\mathcal{C})$ denote the number of edges between clusters (inter-cluster) and $\overline{m}(\overline{\mathcal{C}})$ the number of node pairs not adjacent between different clusters.
The *coverage* metric [12] $cov$ is the fraction of edges contained inside of the clusters and the edges contained in the graph.

$$cov(\mathcal{C}) := \frac{m(\mathcal{C})}{m}$$

The *modularity* metric [13] $mod$ can be viewed as an extension of the coverage metric that punishes trivial clusterings by subtracting a term for the expected coverage for this clustering in a random graph that is constructed to maintain the node degrees.

$$mod(\mathcal{C}) := \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{C \in \mathcal{C}} \left( \sum_{v \in C} deg(v) \right)^2$$

*Performance* [14] perf measures how many node pairs are grouped correctly in the clustering. Two nodes should be grouped together (i.e. are in the same cluster) if they are connected by an edge. This is the number of edges inside clusters $m(\mathcal{C})$. The opposite are all pairs that should not be together as they do not share an edge and are not in the same cluster $\overline{m}(\overline{\mathcal{C}})$. The sum is the number of all pairs correctly classified. This number is normalized with the possible number of pairs in an undirected graph.

$$\text{perf}(\mathcal{C}) := \frac{m(\mathcal{C}) + \overline{m}(\overline{\mathcal{C}})}{\frac{1}{2}n(n-1)}$$

*Conductance* [15] $\phi$ measures the bottleneck of a cut in a graph. The numerator counts the number of edges between the clusters. The denominator enforces a balancing of the cut sizes, as a very small or big cut would result in a bad score.

$$\phi(C) := \frac{m(C, V \setminus C)}{\min(\sum_{v \in C} deg(v), \sum_{v \in V \setminus C} deg(v))}$$

In the *inter-cluster conductance* [16] $icc$ we use the biggest conductance of all clusters as the index. This way the balancing is also accounted for, as a very small/big cluster leads to a bad score which is used in the icc score.

$$icc(\boldsymbol{C}) := 1 - \max_{C \in \boldsymbol{C}} \phi(C)$$

In this formula a property of the inter-cluster conductance score is visible. The worst cluster is the only influence on the score and the rest of the clustering is irrelevant for the metric. If one singleton node that has at least one neighbor is contained in a clustering, the $icc$ is zero.

### 2.1.3 Boosted Trees

Boosted Trees are an ensemble technique in classical machine learning. The base is a decision tree that is fit to the data set. For this, the tree splits the data recursively at feature values. The values are chosen such that the split reduces the error of the model. A prediction can be made by following the tree splits for the features of a sample and taking the mean of the data contained inside the leaf node of the tree. In Figure 2.1 there is a visualization of a decision tree.

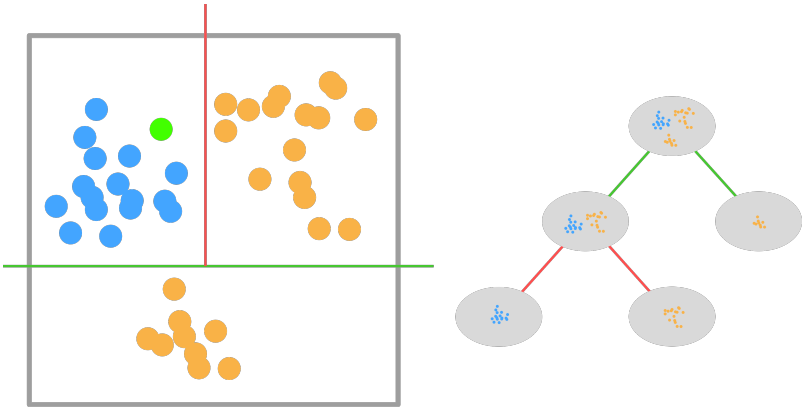

**Figure 2.1:** Idea of a tree-based model for classification.

In the grey box (2D feature space) there are two different classes represented by orange points and blue points. A machine learning model intends to distinguish between those classes based on the position of a point. In this example, we have the green point for classification. The feature space is first divided by the green split. The upper part is divided once more by the red line. On the right is the model that corresponds to this division scheme. The first node contains all data and represents the full space. The green edges represent the decision whether we are above or below the green line. As we end with a pure node (i.e. all data instances belong to the same class) below the line, a deeper subdivision would not provide more insights in this case. For the space above the line, there are two different classes contained. Hence a further subdivision can improve our representation of the data. If we split the data according to the red line, we end up with two pure nodes, each only containing instances of one class. The training error for this model would be zero, as all instances are in pure nodes. The prediction for the green example point in the upper left corner works by traversing the tree. We are above the green line, so we follow the green path to the left of the tree. Also, we are left of the red line, so we again follow the red path to the left node. We end up in a leaf node with only instances of the blue class. The prediction for the green point is that it also belongs to the blue class. Trees, and especially those that were fit until purity, have a huge problem with overfitting. To counter this, there exist multiple ensemble methods using these trees. One of those methods is called *Boosted Trees*. Here a tree is fit to the data such that the training error is not zero, so we do not split until we have pure nodes. Afterward, we fit a new tree and use the errors of the previous tree as the prediction target. This can be repeated multiple times, always training the next tree on the error of the previous trees combined. At prediction time, we calculate the predictions of all trees and add up the results. Additionally to this XGBoost [17] uses a gradient optimization to further improve the model.

### 2.1.4 Neural Networks

*Neural Networks* are a popular approach in modern machine learning. The basic idea of neural networks is to model the functionality of neurons in the human brain. A basic neural network consists of layers of neurons. Through these layers, the input data is passed forward and processed. To fit a neural network to training data, a loss and the corresponding gradients are calculated for the training instances. The loss is minimized by iterative optimization techniques like (stochastic) gradient descent or ADAM [18]. Neural Networks are so powerful, as they are *Universal Approximators* [19]. It means that, if the neural network is chosen sufficiently big, it can approximate any function to any precision in theory. In recent times especially deep neural networks (DNN) have seen big growth. Here we use a simple fully connected neural network (multi-layer perceptron) with a few hidden layers, but there are other neural network architectures like convolutional neural networks (CNN) or recurrent neural networks (RNN). In Figure 2.2 we can see the basic idea of a neuron on the left.

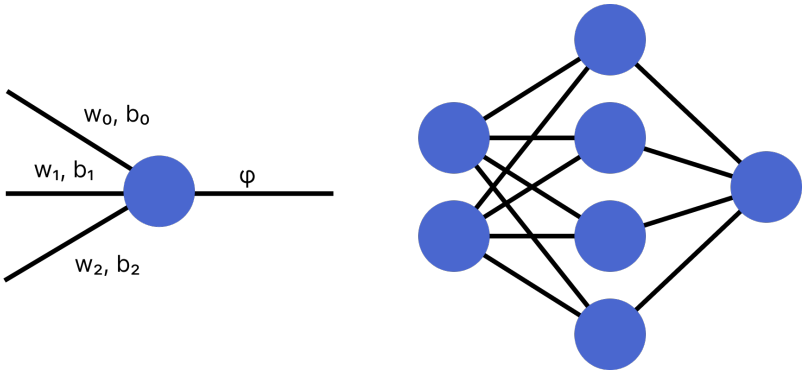


**Figure 2.2:** Single neuron and neural network.

In this example, we get three inputs each weighted by their individual weight $w_i$. Every input is also combined with a bias $b_i$ by addition. Inside a neuron, these inputs are added up. At this point, the function represented by the neuron is still linear. To break this linearity a non-linear activation function $\varphi$ is applied after the sum. The complete function for a neuron with N inputs $x_i$ is $\varphi(\sum_{i=0}^{N-1} w_i \cdot x_i + b_i)$. These neurons are stacked over each other to create a layer. Multiple layers are concatenated to increase the number of free parameters and thus the complexity of functions the network can model. When training a neural network, we have to pick an optimizer that specifies the update rules for the parameters of the network. Although neural networks are all optimized with a policy based on decreasing the loss by gradients, the specific policies differ in complexity and solution quality. We use batches to learn from, as it is faster, converges to better solutions, and is the standard in deep learning. Two examples of policies are stochastic gradient descent (SGD) and ADAM [18]. For SGD a random batch of the training data is sampled and the gradients for this batch are calculated. The weights of the layers are updated by subtracting the gradients with a learning rate to scale the update. In ADAM additional moments of the gradients are used to calculate the change for each weight. The moments make ADAM more adaptive than SGD and lead to faster convergence in many cases.

### 2.1.5 XGBoost vs Neural Networks

Although XGBoost is a very strong machine learning model, it is still restricted by the classical bias-variance tradeoff. There exists a sweet spot for the model to end up in, that minimizes a test error. With this, we can only achieve a certain quality on new data. However as neural networks have shown, by increasing the capabilities of the estimator, we can reach new fields of performance which are not possible with classical machine learning [20] visualized in Figure 2.3.

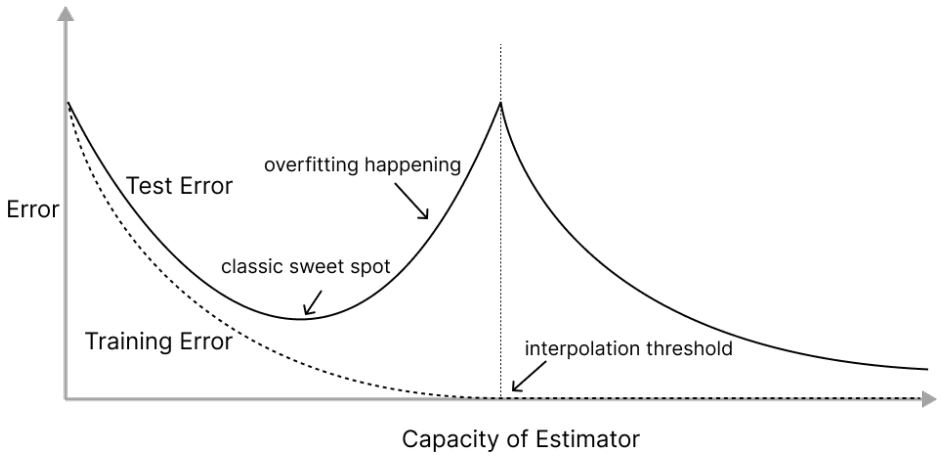


**Figure 2.3:** Modern Bias-Variance trade off.

We want to explore how well the results are for neural networks on our current problem. Another interesting property of neural networks is their continuity. If we encounter data we have not yet seen and it is located between two learned predictions, a tree model needs to decide for one leaf. For neural networks, we can assume that the function has a mostly smooth transition between those points and might produce a prediction that is closer to what would be appropriate for the data in an interpolation-like manner visible in Figure 2.4.

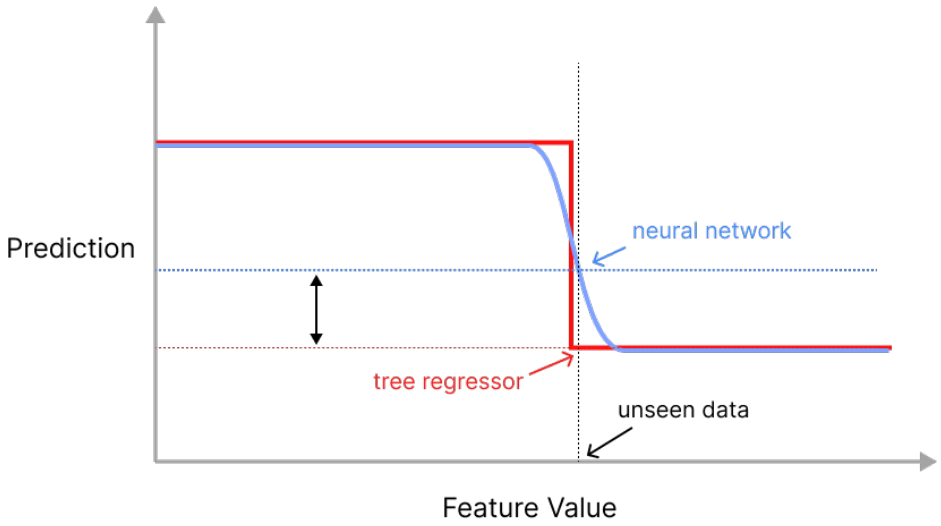


**Figure 2.4:** step vs smooth models.

### 2.1.6 Overfitting and Regularization

When using machine learning techniques in general the problem of overfitting is often encountered. Overfitting describes the situation when an estimator approximates the training data very well, but struggles with the approximation of the true distribution of the problem. This happens as the model remembers some specific information only appearing in the training data distribution. This leads to a very low error while still failing to generalize the problem at hand. As data only consists of a small fraction of all infinitely possible graphs, the data created is subject to a selection bias. As Overfitting is a reappearing problem there are known strategies to reduce it. This is often achieved by a Regularization Term in the loss function used for training the model. XGBoost [17] already uses a regularization term in their library and thus already prevents overfitting, but we also have to address this problem for the neural network. A popular method is the *L2 regularization* of the weight matrices of the layers. Here the Frobenius norm of the weight matrices is added to the loss weighted with a regularization parameter for scaling the influence of the regularization.

$$||A||_F = \sum_{i=1}^{N} \sum_{j=1}^{M} A_{ij}^2$$

This leads to smaller weights in the layers and forces the network to find a more general approximation, as the function becomes smoother. Another less theoretical approach is *Dropout*, where some neurons in a layer are switched off with a given probability. These neurons do not influence the result in the forward pass and are also skipped in the backpropagation step. By disabling some neurons, we create a subnetwork that is trained and can perform the task to some degree. The idea is, that after some iterations there are multiple subnetworks in the network, that work well individually and hopefully together to perform the task. Since a neuron cannot rely on another neuron to be active at any time in training, this reduces codependence and thus makes it harder to overfit.

CHAPTER 3

# Related Work

As graph clustering is a known problem, there exist many different approaches to finding solutions. In the following, we discuss some of these algorithms related to our approach.

## 3.1 Modularity Based

An approach that works well for many problems is to define a metric that measures the quality of a solution and then optimize this metric. If there exists a feasible optimum for the metric, it can be used to provide a solution to the problem. If there exists no known strategy to find the optimum of the metric, heuristics need to be used to approximate it. A very straightforward algorithm that tries to maximize modularity is known as *greedy agglomeration* or *Modularity Maximization* [21]. In this algorithm, we start with a singleton clustering with every node in its cluster. The two clusters are contracted into one cluster that results in the biggest *modularity improvement* for the graph. For this contraction, only clusters connected by an edge are taken into consideration, as a contraction of not incident clusters can never result in a modularity improvement [21]. As every iteration reduces the number of clusters by one, we need to perform $O(n)$ contractions. A naive implementation runs in $O((m + n)n)$ or $O(n^2)$ for a sparse graph. By keeping track of the fraction of edges connecting nodes of clusters and the fraction of edges attached to each cluster we can calculate the modularity change for every possible contraction. We also store the modularity changes for all possible contractions and update them using the two quantities, if we perform a contraction of neighboring clusters. After a contraction, the quantities are updated. By using a binary tree for storing the adjacencies and max-heaps to store and find the modularity changes the complexity of this algorithm drops to $O(md \cdot \log n)$ with $d$ the maximum times a node is contracted (i.e. depth of the dendrogram). Additionally, this clustering process has a *single peak property* [21]. This means that there is exactly one change in the sign of the modularity differences throughout the process. If we find a clustering at some point, after which the modularity can only be reduced, we know that

we are currently at a local optimum and can stop the process. Very powerful techniques in finding solutions for different problems in big graphs are multi-level algorithms. Here the approach is to contract a graph by grouping nodes and edges to create a smaller version of the graph. An algorithm that uses modularity like greedy agglomeration is the *Louvain method* [10]. Here we also start with singletons and work on them in random order. A node is moved into a neighboring cluster such that it produces the biggest modularity improvement. If no further improvement is possible and the local optimum is reached, the clusters are contracted to one new node per cluster. The graph that results from these contractions is significantly smaller than the original graph and allows us to consider bigger decisions than just moving a single node, but moving larger groups. As we cluster the nodes before to maximize modularity, we hope that the basic structure of the graph's groups is maintained. On this contracted graph we repeat the steps before until we contracted the graph as much as possible and we cannot improve the modularity anymore.

## 3.2 Structure Based

Some algorithms rely on the edge density in different regions of the graph to find clusterings. A very simple but still effective method is *label propagation*. This algorithm starts with a cluster for each node. Then the nodes are processed after each other. In one round every node is assigned to the cluster with the best cluster connected to it. This best cluster can be determined in different ways. A common approach is to use the strongest connected cluster (i.e. the cluster with the most edges incident to the current nodes). There are also other strategies for deciding the new assignment [22]. After a few rounds over the nodes, the clustering converges or fluctuates between similar solutions. As each iteration is very cheap this algorithm is suited for very big graphs, with a complexity of $O(n+m)$. Another interesting idea is *ORCA* [23]. Here we look for substructures of high density. If we found all of these structures we contract them and repeat the process. Here we do not rely on metrics for decisions, but still, achieve very good scores in modularity.

## 3.3 Spectral Methods

An often used class of techniques is a spectral analysis of the graph in some form, which relies on eigenvalue decomposition for a solution. Described in [24] is a spectral graph clustering algorithm, that uses the Laplacian matrix of a graph to find a clustering. The Laplacian is composed of the diagonal degree matrix $D$ and the adjacency matrix $A$

$$D_{i,i} = deg(v_i)$$

$$A_{i,j} = \begin{cases} 1 & \text{if } \{i,j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

$$L = D - A$$

Then the k smallest eigenvectors are computed (in terms of the corresponding eigenvalue). Now the $i$th column of a matrix with the eigenvectors as rows is considered the feature vector of the $i$th node. These features are now used to cluster the nodes by the k-means clustering algorithm [25]. Here k initial means are placed randomly in the k-dimensional feature space. Every node is assigned to its nearest mean. Now the means get updated with all features of their assigned nodes. The assignment and update are repeated until convergence. In the end, we receive k clusters for the nodes based on the features and can apply the assignment to the nodes of the graph.

CHAPTER 4

# Learned Agglomerative Graph Clustering

In the following, we present our algorithm and explain each step in detail.

## 4.1 Overall Algorithm

Our algorithm consists of a preprocessing phase followed by a main phase. We start with a preprocessing phase in which supervised training data is prepared and the model is trained. After this step, the model can predict how likely an edge is contained in a cluster. In the main phase of our algorithm, an unseen graph is clustered iteratively. In each iteration, two nodes are contracted to form a new node in the next iteration. In particular, the two nodes to be contracted are connected by the edge with the highest predicted score by our model. This main phase takes $n - 1$ iterations to end with all nodes contracted into one single node. Between the initialization and the one-node end are $n$ possible clusterings. The final solution is now picked by maximizing modularity or another metric. A solution could also be picked by specifying the number of desired clusters. In this case, we can stop early after reaching the desired number of clusters.

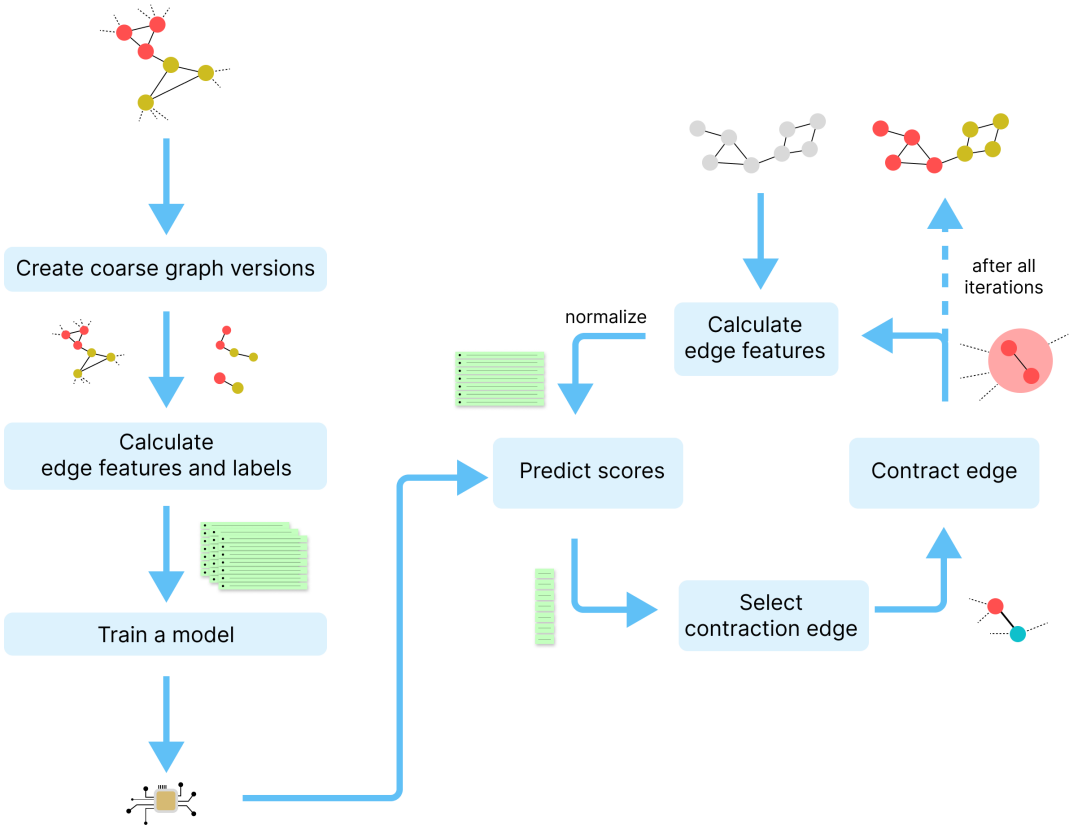


**Figure 4.1:** Overall Algorithm with the preprocessing on the left and main phase on the right.

## 4.2 Preprocessing

Our preprocessing phase aims to produce a model, which is able to predict the probability that an edge should be inside a cluster. Our supervised learning approach is highly dependent on good quality training data. Furthermore, we need meaningful features $X$ for each edge. The function we want the model to learn is

$$\mathbb{P}(\text{ in cluster } | X)$$

### 4.2.1 Training Data

As a base for the training data, we require graphs and a good clustering for each graph. The clustering can be a ground truth clustering from the real world or a clustering previously computed by a high-quality clustering algorithm that might optimize for a specific metric. We use a variety of graphs differing in size and structure to provide enough data for the

model to generalize for unseen graphs with varying properties. We augment the data by adding contracted or *coarsened* versions of the graphs in varying states. These do not change the clusterings in their essence, but only provide more detailed insight into the cases that could appear in the process of clustering a graph. A contracted version of a graph is produced by finding a maximal matching in the graph. The matching is required to respect the cluster borders of the clustering, as no edge between two different clusters can be contained in the matching. All matched nodes are contracted with their corresponding matching partner to a new node per pair. With this, the graph's size is reduced. This creates a new graph with similar properties to the graph in the process of clustering as some nodes are more contracted than others. As the matching happens only within clusters, the information of the clustering is not altered. This coarsening of a graph is repeated until the number of nodes cannot be significantly reduced anymore (see Chapter A), yielding multiple versions of the graph. This augmentation is also necessary to provide training data for some features that change with a contraction. After the augmentation step, we now have a big collection of graphs with a good clustering. For each clustered graph features and labels are extracted for every edge inside the graph. The details of the feature generation are in Section 4.2.2. The prediction target describes if the edge is inside a cluster (1) or if it is connecting two different clusters (0).

### 4.2.2 Edge Features

It is intuitively desirable for edge features to be as general as possible, only relying on the structure of the graph. Consequently, no specific clustering metric is favored. An important part of the edge features is how computationally expensive each feature is and how this could influence the running time of the algorithm. Some features are properties of edges and others are properties of nodes. For node-specific information, the features of both nodes of an edge are contained inside of the feature vector. Before feeding the features to a neural network they are normalized by subtracting the arithmetic average and dividing by the standard deviation of the training data of the network. If the variance in the training data is too large, the magnitude of the gradients for training might explode and lead to numerical errors or huge changes in the weights of the network. Such a normalization is not performed for the tree model. This is not necessary as they perform the same with normalized and unnormalized data.

**Node Degree.** This degree is the number of neighboring nodes a node is connected to.

**Node Count.** This property is the number of nodes $n$ contained in the original graph. This feature is provided for more context about the graph. If a clustering function depends on the size of the graph, this enables the model to learn this dependence.

**Edge Count.** The edge count is the number of edges $m$ contained in the original graph (similar to Node Count).

**Shared Neighbor Count.** This feature counts the neighbors both nodes of an edge have in their neighborhood.

**Clustering Coefficient.** The clustering coefficient [26] describes the local neighborhood of a node in terms of how close it is to being a clique (i.e. how dense it is). It measures how well every node of the neighborhood is connected to every other node of the neighborhood.

$$C_k := \frac{2|\{\{i,j\} \in E : \{i,j\} \in N(v_k)\}|}{|N(v_k)| \cdot (|N(v_k)| - 1)}$$

The denominator is the possible number of undirected edges in the neighborhood of the node $v_k \in V$. As the graph is undirected, each edge that exists needs to be accounted for twice.

**Graphlet Counts.** A graphlet (or sometimes called graph motif) is a simple substructure inside of a graph. Structures for graphlets can be enumerated by their number of nodes and edges. Some simple examples are triangles consisting of 3 nodes all connected or a 4-cycle build from 4 nodes connected in such a way, that a cycle is formed with no other edges crossing the center of the cycle. Although there are infinitely many graphlets, we restricted us to graphlets of order 4 (consisting of 4 nodes) and below due to computation time and practicality, as the number of possible graphlets grows exponentially with the order. The graphlets for one and two nodes are trivial as it is just the node itself or two nodes connected by an edge.

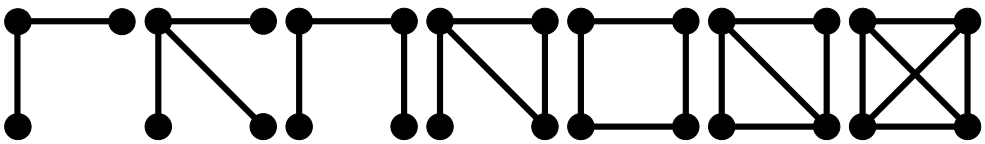

**Figure 4.2:** Used graphlets from left to right: 2_star, 3_star, 4_path, 4_tailed_triangle, 4_cycle, 4_chordal_cycle, 4_clique from the PGD documentation [27].

For calculating the graphlets we use the PGD library [27]. The calculation of graphlets is expensive as there are many possible graphlets and we want to identify all that appear inside of the graph. If we search for graphlets naively we would have to check for every

edge if it is contained in each specific graphlet and if so, how often. This would require us to traverse the local area around each edge until all possibilities for the graphlet have been checked. To reduce the computations, the authors of PGD describe relationships between different graphlets and how counts of one graphlet can be used to derive the counts for other graphlets in constant time or with significantly fewer computations than before. Additionally, the individual calculations are mostly independent of each other, so with concurrency, the running time is further reduced.

**Node Weight.** The node weight gives information about the number of nodes a contracted node consists of. It is a helpful feature, as it makes the nodes transparent.

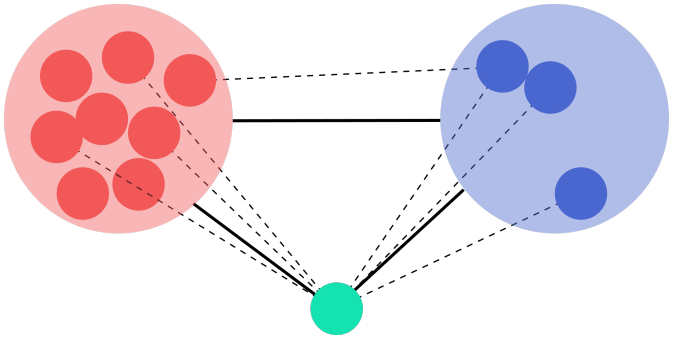

**Figure 4.3:** Example of differing node weights.

In the example in Figure 4.3, we can see two contracted nodes, the red one containing more nodes than the blue one. The green node is connected in the same way to both nodes. The original edges are visualized with dotted lines. If we want to add the green node to one of the other nodes in a natural way, we can see an important difference. As the number of edges connecting the green node with each other node is the same, that means that it is relatively more connected to the blue node than the red node. In the red node, the green node is not connected to many nodes inside, meaning it is most likely not as good in the red cluster as it would be in the blue one. Another aspect of this feature is the balancing of cluster sizes. If our data has balanced clusters, this means that an edge adjacent to a node with a very high weight can have a smaller probability to add more nodes further on compared to a node that is not as big. If the model can capture this property we achieve some sort of balancing of the clusters from the data for free.

**Edge Weight.** The weight of the edge is a strong indicator, of how closely related two nodes are. This becomes especially important after some contractions as it indicates the actual number of edges between the two nodes.

**Internal Edge Weight.** The internal edge weight counts the number of edges that are completely contained inside of a contracted node. This shows how dense the contracted node/cluster currently is.

### 4.2.3 Training the Model

With the previously created data, we fit a model (we only consider boosted trees and neural networks, but any supervised learning technique is possible). Training or fitting a model is done by minimizing the *loss* of the model that measures the magnitude of errors of the model. We use two variants of the *mean squared error* as we want to estimate a regression value for the edge to be inside of a cluster as a smooth function. For this, using two neurons on the last layer with a softmax activation function and the binary cross-entropy loss would also be possible. The default loss for XGBoost is the *root mean squared error*. For predictions $\widetilde{y_i}$ and true values $y_i$ it is defined as:

$$RMSE(y, \widetilde{y}) = \sqrt{\frac{1}{N}\sum_i (\widetilde{y_i} - y_i)^2}$$

For the neural network, we use the pure MSE loss without the root.

**Algorithm 1** Model Creation.

**Data:** Graphs $G_i = (V_i, E_i)$, Clusterings $\mathcal{C}_i \forall i \in \{1, \dots, I\}$, Feature set $F$
**Result:** Model $M$
$X \leftarrow$ empty list
$Y \leftarrow$ empty list
**for** $i = 1, \dots, I$ **do**
  $x \leftarrow$ CreateEdgeFeatures($G_i, F$)
  $y \leftarrow$ CreateEdgeLabels($G_i, \mathcal{C}_i$)
  $X$.appendItems($x$)
  $Y$.appendItems($y$)
  $j \leftarrow 0$
  $G_{i,0} \leftarrow G_i, \mathcal{C}_{i,0} \leftarrow \mathcal{C}_i$
  **while** $G_{i,j}$ *not coarse enough* **do**
    $G_{i,j+1}, \mathcal{C}_{i,j+1} =$ ContractGraph($G_{i,j}, \mathcal{C}_{i,j}$)
    $x \leftarrow$ CreateEdgeFeatures($G_{i,j+1}$)
    $y \leftarrow$ CreateEdgeLabels($G_{i,j+1}, \mathcal{C}_{i,j+1}$)
    $X$.appendItems($x$)
    $Y$.appendItems($y$)
    $j \leftarrow j + 1$
  $M \leftarrow$ FitModel($X, Y$)

## 4.3 Main Phase: Clustering Process

### 4.3.1 Overall Procedure

In the main phase, we contract the graph until all nodes are contained inside one node or a specific criterion is reached. In every iteration, features are calculated for all edges. With these features, a score is calculated with a trained model. Based on these scores an edge is selected and then contracted.

### 4.3.2 Feature Calculation

In Table 4.1 are the costs for calculating the individual features in one iteration.

$T_{max} :=$ max. number of incident triangles to an edge
$S_{max} :=$ max. number of incident stars to an edge

| Feature Name | costs |
|---|---|
| Degree | $O(m)$ |
| Node count | $O(m)$ |
| Edge count | $O(m)$ |
| Shared neighbor count | $O(m \cdot \Delta)$ |
| Clustering coefficient | $O(m \cdot \Delta)$ |
| Graphlet counts | $O\left(m \cdot \Delta \cdot \begin{cases} T_{max} \text{ if } T_{max} \ll \Delta \\ S_{max} \text{ if } S_{max} \leq \Delta \end{cases}\right)$ |
| Node weight | $O(m)$ |
| Edge weight | $O(m)$ |
| Internal edge weight | $O(m)$ |

**Table 4.1:** Feature generation costs in one iteration for all edges.

The node/edge counts, weights, and internal edge weights are stored and updated. With this, the calculation is one memory access for each edge. For node-related features, both adjacent node features need to be accessed hence two memory accesses are performed per edge. The degree of a node is the size of its neighborhood and is calculated twice for each edge in $O(1)$, and thus the cost for the degree is $O(m)$ as we have to provide the degree for every edge. The cost of computing the graphlet counts is derived in [27]. Distributing the graphlets on each edge ($O(m)$) does not change the complexity as the computation of the counts dominates the overall costs.

### 4.3.3 Edge Selection

The model is provided with the generated features and produces a regression value for every edge. The higher the prediction is, the more the model predicts the nodes connected by the edge to be in the same cluster. The edge with the highest score is selected for contraction. If multiple edges have the same maximal value one random edge is picked as the model predicts the same probability for each.

### 4.3.4 Contraction

In the contraction, we merge two nodes into one and update the properties of the new node and edges as defined in Chapter 2.

## 4.4 Theoretical Running Time

To initialize our data structures we have to insert all edges for every node. The insertion of one edge for a node happens in $O(1)$, as the sizes of all arrays are the node degrees and thus are known. This creates an initialization cost of $O(m)$. As a contraction reduces the number of nodes by one per iteration, we perform $O(n)$ iterations to end with all nodes contracted into one cluster. In each iteration after the prediction, the maximum score edge and contraction nodes $j, k$ are found in $O(m)$. Then the contraction is performed. We process all edges of $k$ leading to a node $l$ by updating the $j$th row's entry for $l$ and the $l$th row's entry for $j$ if an edge to $l$ already exists for $j$. If not, in both rows a new edge is inserted leading to a cost for both cases of $O(deg(j) + deg(k))$ as we have to perform the insertions for all edges and we assume the worst case of no shared neighbors such that the degrees sum up. This is similar to [21] and in the worst case $O(m)$ insertions happen. The contraction hierarchy forms a dendrogram with depth $d$. With this, we have at most $O(md)$ insertions in the clustering process. All edges connected to $k$ are removed for all neighbors in $O(deg(k))$ and the $k$th row is removed in $O(1)$. The edge removal is thus also in $O(m)$. The algorithm without feature calculation has a complexity of $O(n \cdot m + md)$. The depth of the dendrogram is limited by the number of nodes $n$, thus if $d = n$ the complexity is $O(n \cdot m)$. As our algorithm does not need the modularity improvement for all possible contractions, a linear data structure can be used. With this, insertions happen in $O(1)$ instead of $O(\log n)$, as we do not sort by modularity improvement.
An issue with this algorithm is that we recalculate rather expensive features like graphlet counts multiple times from scratch and as we only contract two nodes at a time, the graph does not shrink significantly for the next calculation. This heavily limits the graph size we can calculate clusters for. To address these problems we can use an update policy for the clustering process. A feature falls in either of 3 categories regarding changes due to contractions.

1. Changes globally (e.g. node count)

2. Changes for the contracted nodes and edges (e.g. internal edge weight)

3. Changes for local area (e.g. graphlet counts)

The first two categories are the easiest to address: global feature changes are often very simple and in most cases, no change or easy operations like subtraction of 1 are required for the update. Another convenient property is that we do not care, which edge we work on, as all edges are affected in the same way. Changes that only affect the contracted nodes and edges adjacent are also very simple to update. We can update the weights of the new node and at most $2\Delta$ edges. The new feature values are often just sums of the contracted nodes or edges' features. More of a challenge are features, that capture a wider field of the graph like graphlet counts. If we contract two nodes, the graphlet counts for all edges in a distance depending on the graphlet diameter change. For example, the 2-star graphlet has a smaller area than the 4-path graphlet, where counts can be influenced by a contraction. Our solution for these features is to extract an induced subgraph of the local area by performing a breadth-first search up to the point where the area of effect ends. Only on this induced subgraph the relevant features that can change are calculated from scratch and updated in the feature data. All the data that belongs to edges outside of the affected area remain untouched. This naive approach turned out to be a bit slower than just recalculating the features from scratch. Update policies for each graphlet and different data structures could speed up the feature generation potentially.

## 4.5 Memory Consumption

Using big graphs results in many coarsened versions and also in many feature vectors for each edge. If we want to train a tree-based model (like XGBoost), we need a lot of memory, as the data needs to be completely accessible in memory and additional structures are also needed for training. This is not as big of a problem for a neural network, as they are trained with batches of data that are loaded into memory sequentially. However, more training data increases the training time.

## 4.6 Model Switching

By using different hyperparameters or estimators, multiple models can be produced out of the same training data. To improve the quality of the solution different models can be used at different times of the clustering process. We allow switching from an XGBoost-based model to a neural network at a specified time in the process. The algorithm stays the same, however, the predicted scores of the models might differ and thus lead to different results when switching between models.

## 4.7 Data Structures

We use a dynamic array for every node, that contains an entry for every edge that is connected to this node. Additionally, every array stores a map. With this, we can find an element in the array in $O(1)$. The removal of entries happens in $O(1)$, as we can swap the last element with the element we want to remove ($O(1)$) and decrease the size of the array by one ($O(1)$).

**Algorithm 2** Learned Agglomerative Graph Clustering.

**Data:** Graph $G = (V, E)$, Model $M$
**Result:** Clustering $\hat{\mathcal{C}}$, Modularity $\hat{Q}$
$n \leftarrow |V|, m \leftarrow |E|$
`// store original graph`
$G' \leftarrow G$
$\mathcal{C} \leftarrow$ Array$(1, \ldots, n)$
$NW(v) \leftarrow 1, iew(v) \leftarrow 0 \forall v \in V$
$EW(e) \leftarrow 1 \forall e \in E$
$Q \leftarrow$ CalculateModularity$(G', \mathcal{C})$
$\hat{Q} \leftarrow Q$
$\hat{\mathcal{C}} \leftarrow \mathcal{C}$
**for** $i = 1, \ldots, n-1$ **do**
  $X \leftarrow$ CreateEdgeFeatures$(G, NW, EW, iew)$
  $Y \leftarrow$ PredictEdgeScores$(X, M)$
  $j \leftarrow \max_{j'=1,\ldots,m} Y[j']$
  $\{u, v\} \leftarrow E[j]$
  $NW(u) \leftarrow NW(u) + NW(v)$
  $iew(u) \leftarrow iew(u) + iew(v) + EW(\{u, v\})$
  `// remove the contracted edge`
  $E \leftarrow E \setminus \{\{u, v\}\}$
  `// process outgoing edges of the contracted nodes`
  **foreach** $\{v, v'\} \in E$ **do**
    **if** $\{u, v'\} \in E$ **then**
      $EW(\{u, v'\}) \leftarrow EW(\{u, v'\}) + EW(\{v, v'\})$
    **else**
      $E \leftarrow E \cup \{u, v'\}$
      $EW(\{u, v'\}) \leftarrow EW(\{v, v'\})$
    $E \leftarrow E \setminus \{\{\{v, v'\}\}\}$
  $V \leftarrow V \setminus \{v\}$
  $m \leftarrow |E|$
  $\mathcal{C}[v] \leftarrow \mathcal{C}[u]$
  **foreach** $v'$ *contained in* $v$ **do**
    $\mathcal{C}[v'] \leftarrow \mathcal{C}[u]$
  $Q \leftarrow$ CalculateModularity$(G', \mathcal{C})$
  **if** $Q > \hat{Q}$ **then**
    $\hat{Q} \leftarrow Q$
    $\hat{\mathcal{C}} \leftarrow \mathcal{C}$

CHAPTER 5

# Experimental Evaluation

To construct this algorithm and the feature sets, we conducted several experiments.

## 5.1 Methodology

We implemented our algorithm in the KaHIP [28] framework in C++ and compiled it with g++ 11.2.0 using loop unrolling (-funroll-loops) and no additional optimization turned on. We also implemented the greedy agglomeration algorithm in this framework as a base for our algorithm. VieClus was compiled from the latest version (Apr 7 13:39:26 2021) with the default configuration and above compiler. All experiments were performed on the same machine with a four-core Intel i5-6600K running at 3.50 GHz with a boost frequency of 3.90 GHz, 24 GB main memory, and 6 MB L2-Cache. For the neural networks, CUDA 11.7 with cuDNN 7.2.1 was used on an RTX 2060 with 6 GB memory. The machine runs Ubuntu GNU/Linux 22.04.1 LTS and Linux kernel version 5.15.0-47-generic. Evaluations on the smaller graphs were performed three times, due to the long runtime we ran the bigger graphs only once. We got the graphs partly from the Walshaw benchmarks [29], the DIMACS10 challenge [30] and individual sources [31]. The training data consist of the Training Graphs in Table 5.1 with a clustering created by VieClus. The graphs are selected by their size. We calculate coarsened versions with the global path matching algorithm [32] until the node count stops decreasing noticeably. As we must calculate features for every graph version, using very large graphs becomes less feasible, so only graphs up to a specific size are used. Also as the clustering process is computationally expensive only smaller graphs are suited. These are not contained in the training data. We use modularity as our main evaluation metric to improve our algorithm throughout experiments. For this, both the final modularity for the best clustering and the modularity development are taken into consideration. We first compare different feature combinations with an XGBoost [17] model with default hyperparameters and five training iterations. Afterward, we do the same for a neural network implemented with Libtorch, the C++ API of PyTorch [33] version

1.11.0 trained with ADAM [18], a learning rate of 1e-5 and an L2 regularization with a strength of 2e-4. The model consists of 4 layers with the first layer being the feature input size and the last layer of size one for the prediction value. The layers in between both have 400 neurons. The network uses the ReLU activation [34] after all linear layers except the last one, as we want a regression value. The network is trained for two epochs. Following we explore if preventing overfitting of the neural network improves the solution quality. After noticing differences between the two approaches, we try to switch between the models halfway through the clustering process. Then, we try to replicate clusterings for specific graphs to explore the adaptability of the model.

| Evaluation Graphs | | | |
|---|---|---|---|
| Graph | n | m | Type |
| chesapeake | 39 | 340 | Ecosystem Network |
| dolphins | 62 | 318 | Social |
| lesmis | 77 | 508 | Coappearance |
| adjnoun | 112 | 850 | |
| football | 115 | 1,226 | |
| jazz | 198 | 5,484 | Social |
| celegansneural | 297 | 4,296 | Neural Network |
| celegans_metabolic | 453 | 4,050 | Genes |
| email | 1,133 | 10,902 | Social |
| netscience | 1,589 | 5,484 | Coauthorships |
| add20 | 2,395 | 14,924 | |
| cora | 2,708 | 10,556 | Citation |
| data | 2,851 | 30,186 | |
| 3elt | 4,720 | 27,444 | |
| uk | 4,824 | 13,674 | Roads |
| power | 4,941 | 13,188 | Network |
| add32 | 4,960 | 18,924 | |
| hep-th | 8,361 | 31,502 | Coauthorship |
| PGPgiantcompo | 10,680 | 48,632 | Social |
| wing_nodal | 10,937 | 150,976 | |
| astro-ph | 16,706 | 242,502 | Coauthorship |
| cond-mat | 16,726 | 95,188 | Coauthorship |
| as-22july06 | 22,963 | 96,872 | Network |
| cond-mat-2003 | 31,163 | 240,058 | Coauthorship |
| cond-mat-2005 | 40,421 | 351,382 | Coauthorship |
| t60k | 60,005 | 178,880 | 2D nodal graph |

**Table 5.1:** Evaluation Graphs for experiments.

| Training Graphs | | | |
|---|---|---|---|
| Graph | n | m | Type |
| 3-cluster | 17 | 27 | Artificial |
| zachary | 34 | 78 | Social |
| polbooks | 105 | 441 | Network Co.-Purch. |
| whitaker3 | 9,800 | 28,989 | Mesh |
| crack | 10,240 | 30,380 | 2D nodal graph |
| fe_4elt2 | 11,143 | 32,818 | |
| 4elt | 15,606 | 45,878 | 2D nodal graph |
| fe_sphere | 16,386 | 49,152 | |
| cti | 16,840 | 48,232 | |
| memplus | 17,758 | 54,196 | 2D nodal graph |
| cs4 | 22,499 | 43,858 | |
| 598a | 110,971 | 741,934 | |
| luxembourg | 114,599 | 119,666 | Roads |
| citationCiteseer | 268,495 | 1,156,647 | Citation |
| belgium | 1,441,295 | 1,549,970 | Roads |
| netherland | 2,216,688 | 2,441,238 | Roads |
| germany | 11,548,845 | 12,369,181 | Roads |
| Optional Training Graph | | | |
| fe_body | 45,087 | 163,734 | |

**Table 5.2:** Training Graphs for experiments.

## 5.2 Feature Development

We provide different sets of features to approximate the local structure around an edge and let the machine learning model identify the meaningful ones by itself. We perform these experiments first with the tree model [17], as it is not as dependent on good hyperparameters as neural networks.

### 5.2.1 Initial/Minimal Feature Set

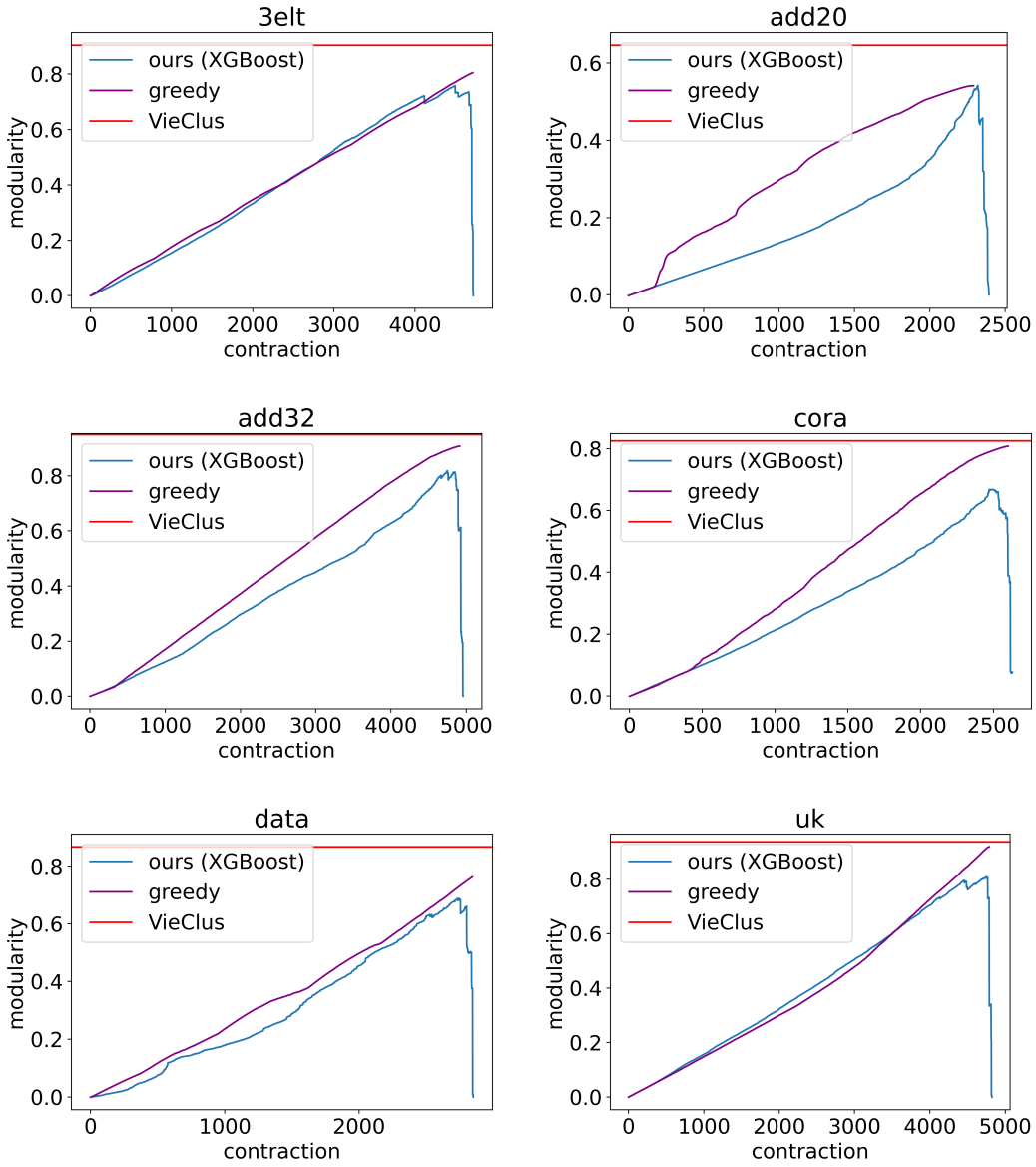


**Figure 5.1:** Modularity Development on different graphs with the minimal feature set.

The features used initially are the *node degrees*, the *shared neighbor count*, the *clustering coefficient* and the *graphlet counts*. Figure 5.1 shows in every plot the change in modularity for a specific graph. The x-axis is the number of contractions performed to reach this point in time. This axis always starts at 0 and ends at the number of nodes contained inside of a graph after all possible nodes are contracted. The y-axis is a metric for the clustering

process for insights into the quality of the current model. In these plots, it is the modularity of the clustering at the current contraction. To provide some reference, two other clustering results are provided. The greedy agglomeration is plotted the same way as our results as a purple line. Since greedy agglomeration can rely on the single peak property [21], the line stops before all clusters are contracted into one and there is no drop after the peak as the other plots have. Also, a clustering produced by the VieClus algorithm is used for comparison to see the possible potential. The modularity of this solution is the horizontal red line. For our algorithm, the most relevant part is the highest point in the process, as we can just record this solution and keep on contracting the clusters together. If we find a better solution than before, we update the current best. In the end, we just use the best solution (the maximum) as a result of the clustering. In contrast to the greedy agglomeration, we do not have a single peak of the modularity curve guaranteed (see data or uk plot in Figure 5.1). This makes it impossible to stop early as a better result could be achieved after a small peak with a little drop. If we have constraints on how many clusters $k$ we want to end with, we can also just use the clustering after $n - k$ contractions.

We can already see from these plots, that we can produce non-trivial clusters (Figure 5.10, random contractions with mean modularity around 0.2), and even more than that. In the instance add20 our clustering is a tiny bit better than the results of the greedy agglomeration (greedy: 0.5411, ours: 0.541927). Another very interesting observation here is in the uk plot. Greedy agglomeration is specifically targeted to maximize modularity by always picking the local best modularity improvement available. However, our algorithm can find contractions in such a way, that we end up with higher modularity in the process between contraction 1000 and 3500 than the greedy agglomeration. An important reminder at this point is, that our algorithm does not know the change of modularity that is connected to a contraction of nodes. It only has seen data of clusterings with high modularity. This means that by picking a different edge for contraction than the maximal modularity improvement, we can achieve at least temporary better results. However, we can notice, that the features provided with the minimal feature set are only capturing the graph at surface level, ignoring any information kept inside of nodes and edges.

### 5.2.2 Node Weight

If we add the node weight to the feature set, we see a small improvement. For some graphs (3elt, add32, uk) the maximum modularity improves visibly in Figure 5.2. For other graphs, it seems to have almost no influence. The average modularity with node weights (0.724067) is higher than without (0.707156).

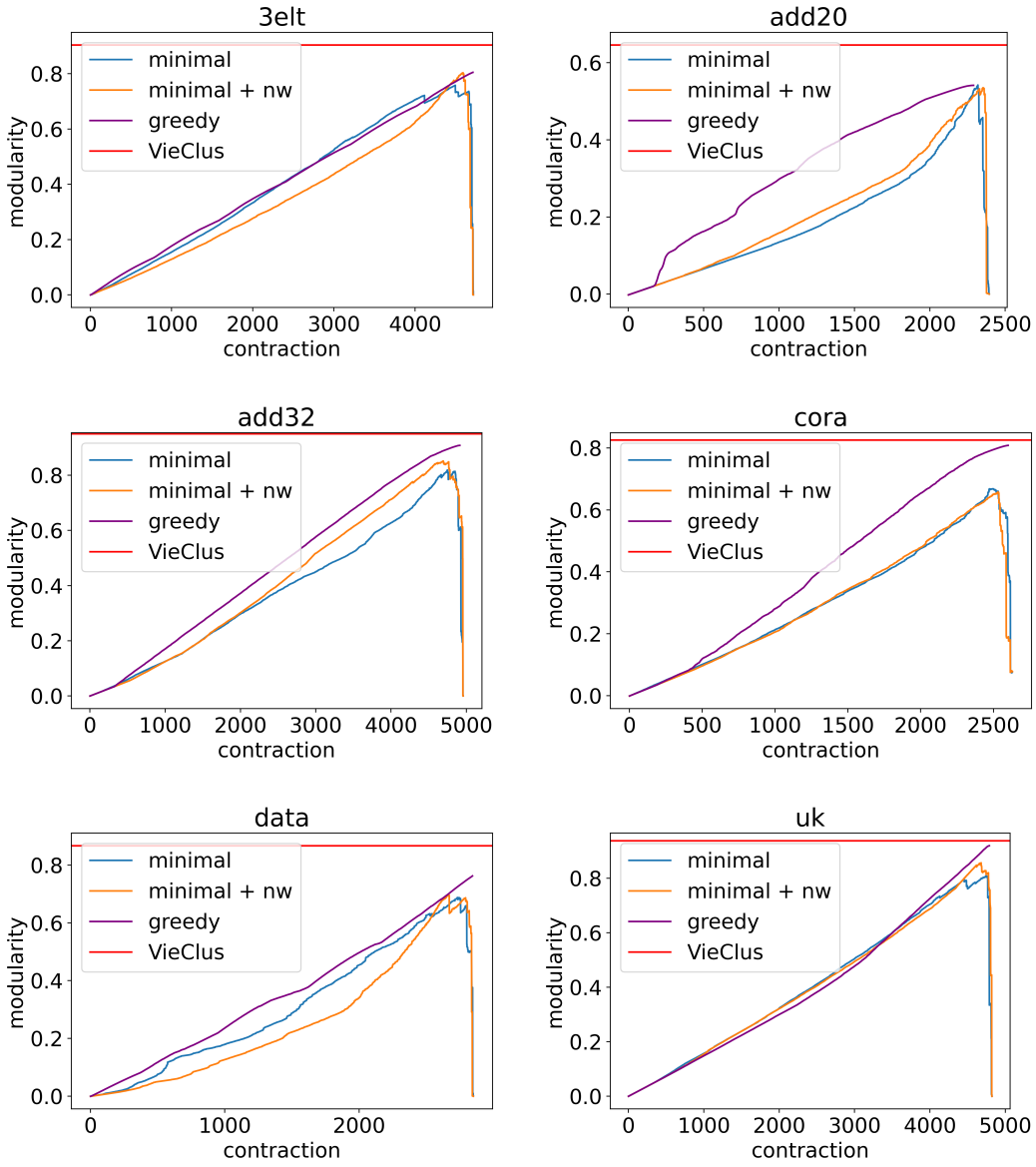


**Figure 5.2:** Modularity Development on different graphs with minimal feature set vs node weights added.

### 5.2.3 Counts and more Weights

After making the weight of nodes accessible and with that the nodes that are otherwise hidden within a contracted node, the next idea is to also make the edges transparent. This includes both edges that are still in the graph structure accessible but also edges that are locked inside of nodes because they are involved in an earlier contraction. For the first, every edge has a new weight feature. For the second category of transparency, every node

receives an internal edge weight and thus an edge has access to the internal edge weights of both nodes it is connecting. Another piece of information that can be interesting is the number of present nodes and edges in the original graph. If there exists a dependence of a clustering function on the size of the graph, the model cannot use it for the clustering process. Therefore, we add this information and can see the results for different configurations in Figure 5.3.

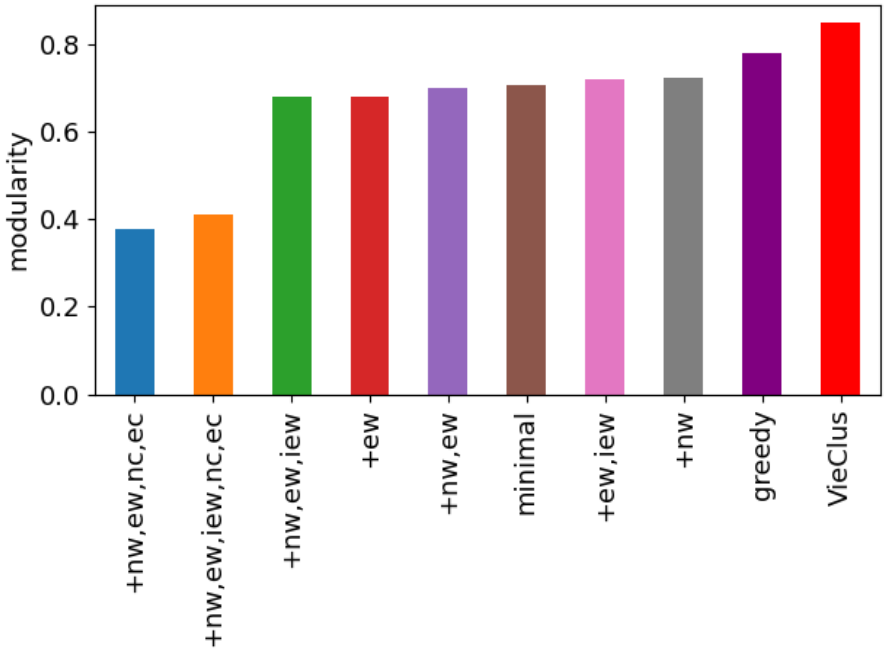


**Figure 5.3:** Geometric mean modularity for different feature sets vs greedy agglomeration and VieClus.

After adding different features, we observe that more features do not equal better results. If we add the weight features separately (nw: node weight, ew: edge weight, and iew: internal edge weight), we see a small improvement, but when combining the node and edge weight features the results are worse than with just the minimal feature set. Also, after adding the counts (nc: node count, ec: edge count) we see a significant drop in the modularity.
In Figure 5.4 we see the differences in the modularity development for bigger feature sets using the node count. In the beginning, they all perform better or equally good as the minimal feature set in terms of modularity. The initial better results might suggest, that the additional information about the graph is helpful. As the contractions progress, the models start to behave similarly to a model picking random edges in some graphs. Although the feature set models with more information start better than the minimal feature set model, they are in some instances unable to find edges, that produce a good clustering later on. For the minimal model, the maximum spike of modularity seems to be very concentrated, as

the curve rises, stays there for a small number of contractions, and drops at the end. For the feature set model using internal edge weights and counts there exists no such sharp point of highest modularity, but the modularity seems to plateau for longer.

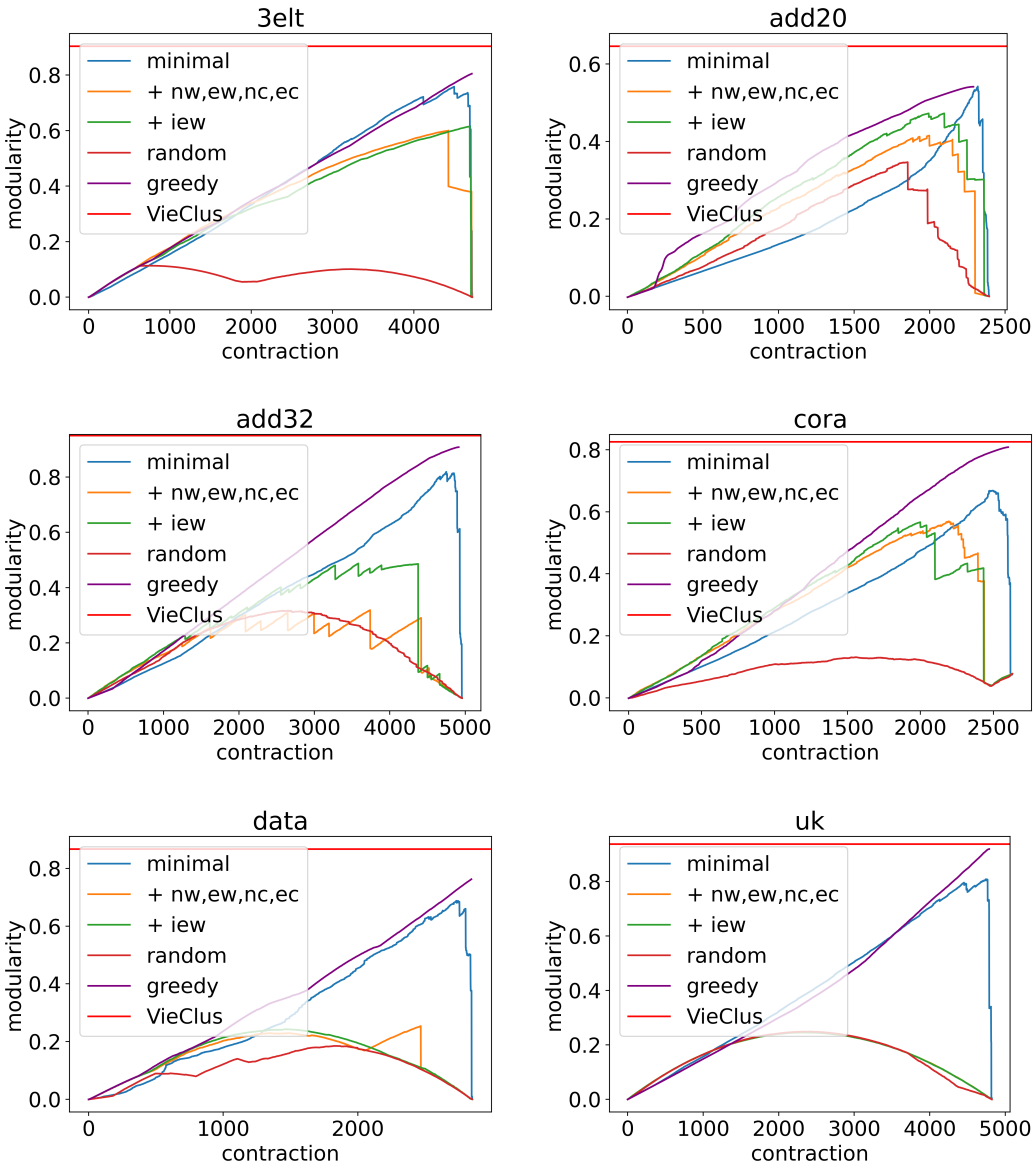


**Figure 5.4:** Modularity Development on different graphs and different feature combinations using counts.

As we increase the feature dimensions, this significantly increases the feature space and leads to the fact, that we usually need more data to train a model that works well with this feature set. We suspect that the problem lies in the amount of training data, as the model

seems to perform well in general, but struggles with a specific contraction stage of a graph in some cases. To support this argument and to show that it is not a general restriction imposed by the new features, we add one graph instance (fe_body) to the training set, augment the data with the coarsened graphs, and train a new model on this data set.

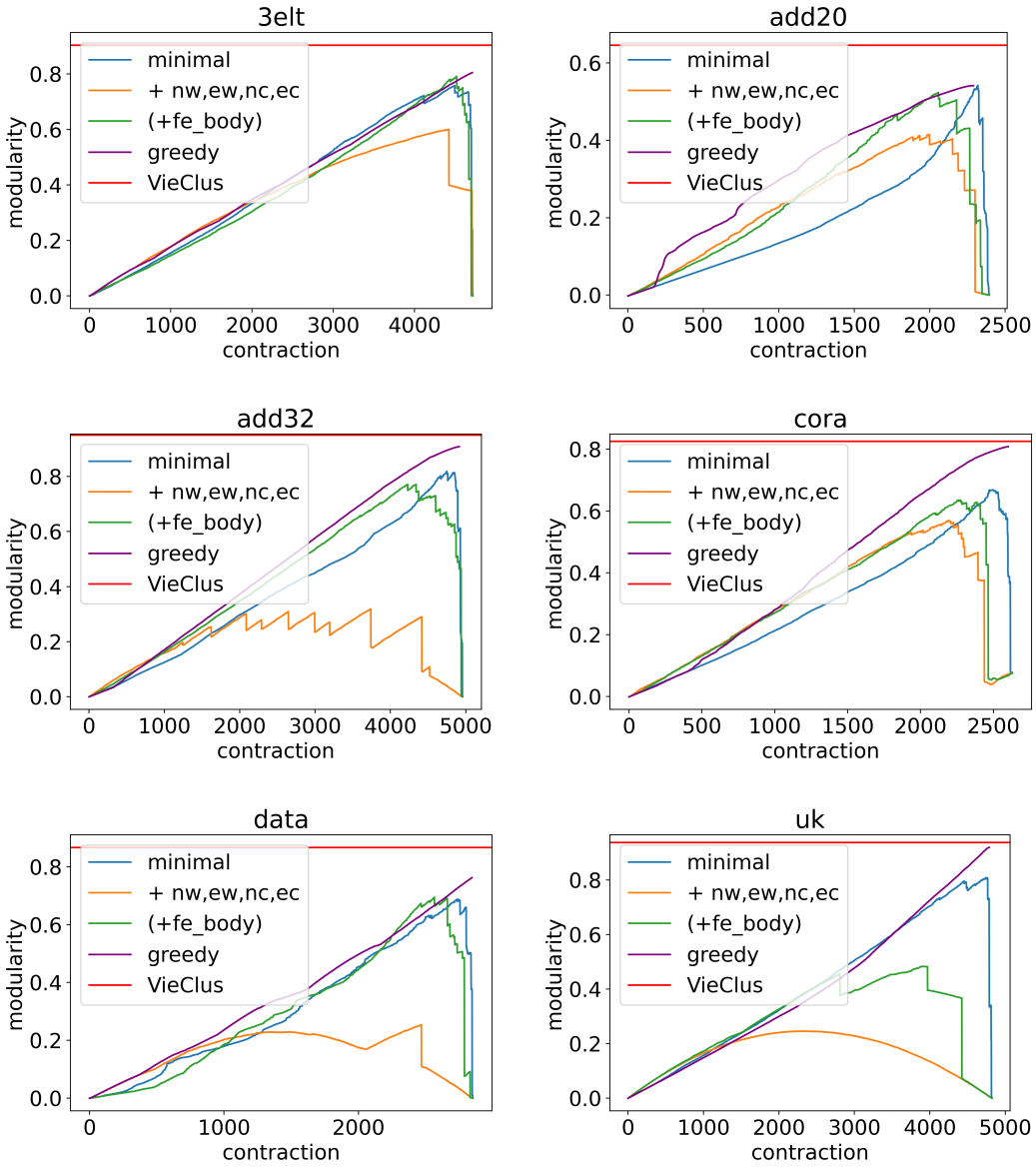


**Figure 5.5:** Modularity Development on different graphs after adding fe_body graph to training graphs for the feature set without internal edge weights.

We can see in Figure 5.5 that there is a huge improvement in the achieved modularity. Every instance is clustered with higher modularity than before. Also, the random-like behavior

ceases to happen. In the case of the data graph, it is also slightly better than the minimal feature set. Only in the uk graph, the progression remains partially similar to the random model. If this trend continues, this also validates the intuition that the result should improve when providing more training data with more variety.

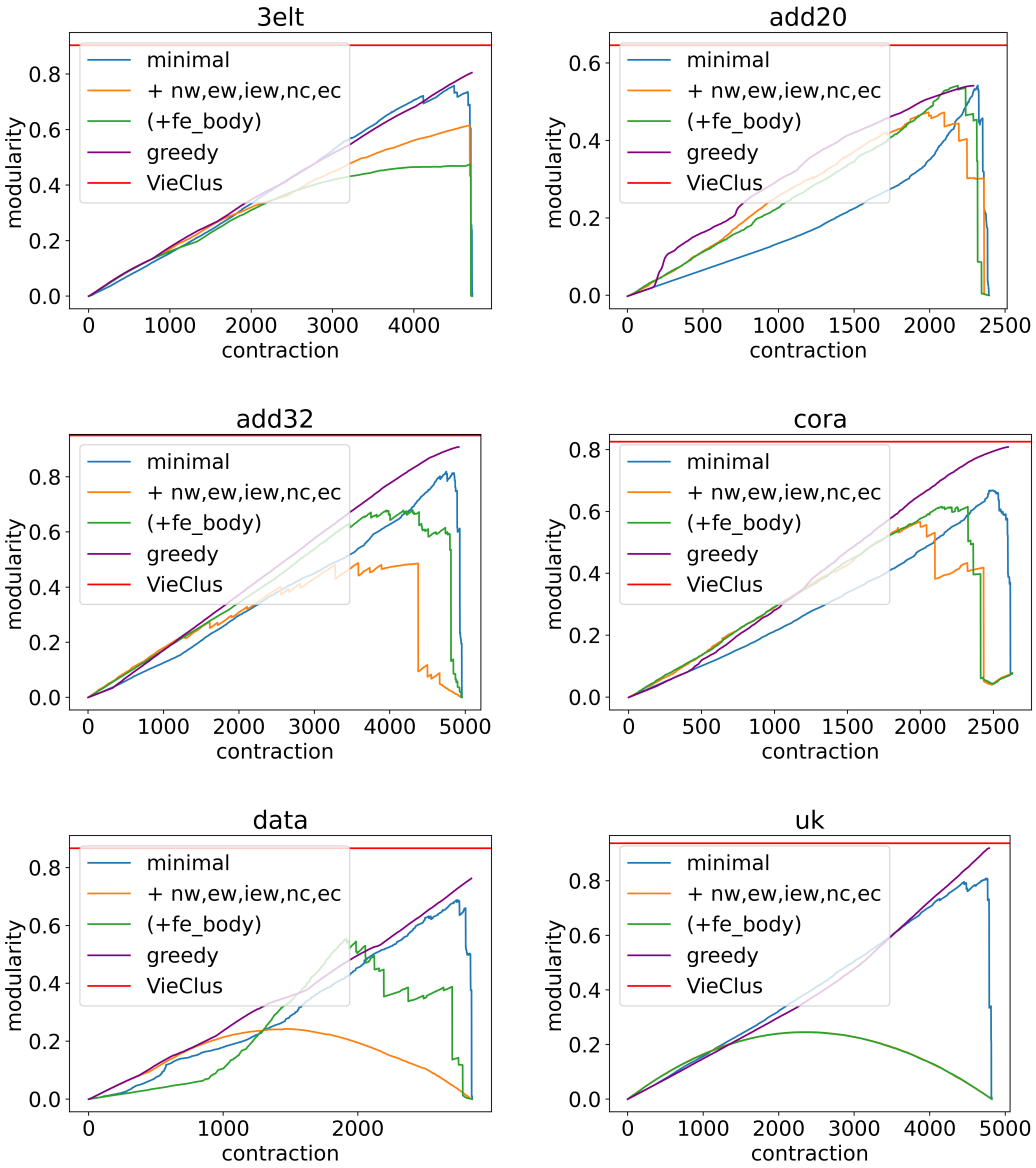


**Figure 5.6:** Modularity Development on different graphs after adding fe_body graph to training graphs for the feature set with internal edge weights.

In Figure 5.6 we see the effect of adding data to the training set when also using the internal edge feature. For most graphs, it is a significant improvement. For the uk graph, the addi-

tional data does not change anything and for the 3elt graph, it results in lower modularity, as the modularity grows even slower. A likely explanation is a core difference in the graph structures.
So far the best feature set with fixed training data we have is using the minimal set with the node weight. There is also a strong indication, that adding the node and edge count requires more training data with more variety in the data to produce good results, maybe yielding even better results than the other feature sets with a sufficiently large data set.

## 5.3 Neural Network

We can perform the same experiments as we did with the tree model with a neural network as our estimator. The geometric mean of achieved modularities for the neural network for different feature sets is visible in Figure 5.7.

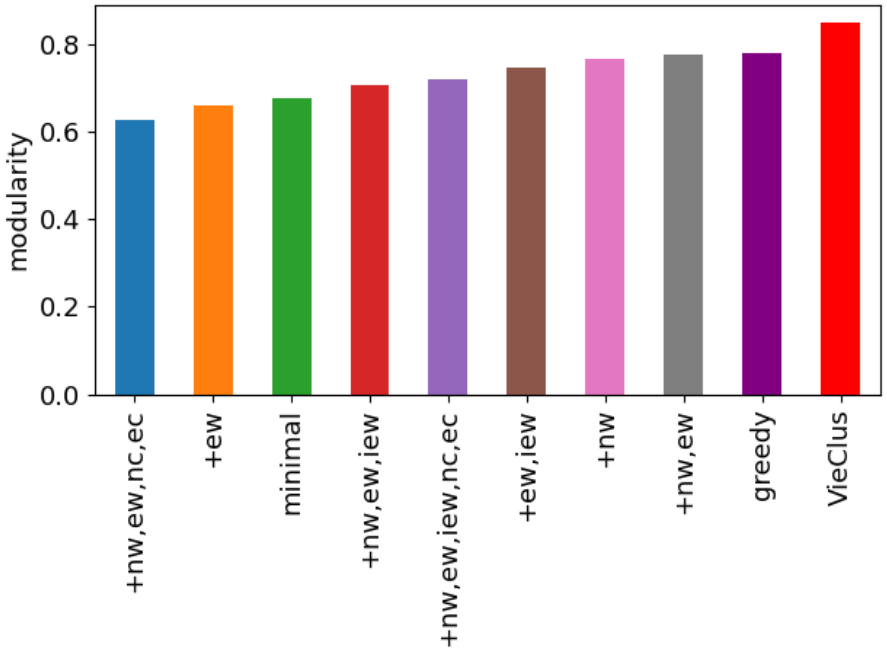


**Figure 5.7:** Geometric mean modularity for different feature sets vs greedy agglomeration and VieClus.

Right from the start, we can see an interesting difference for the neural network. In contrast to XGBoost, the difference between our best feature set model and the greedy agglomeration is now almost zero. Also, the results for the feature sets with node and edge count are significantly better than for XGBoost. The worst result for XGBoost is below 0.5 and for

the neural network easily exceeds 0.5. Also, the feature set with the counts and the internal edge weight is even better than the minimal feature set.

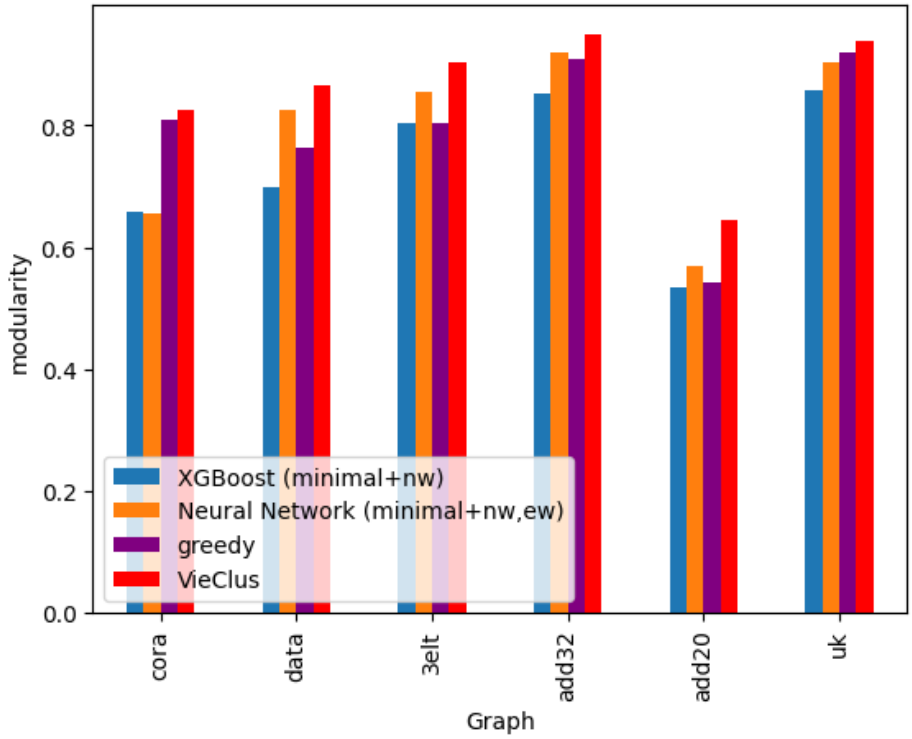


**Figure 5.8:** Modularity for XGBoost and Neural Network vs greedy agglomeration and VieClus.

We can see in Figure 5.8 that except for the first instance all clusterings calculated with the neural network are better than the ones created with XGBoost. For 4 of 6 instances, it is also better than using the greedy agglomeration algorithm. However, none of the clusterings were able to beat VieClus. From here on the experiments are conducted on the neural network, as it yields better results.

## 5.4 Dropping Graphlets

Graphlet counts give a detailed representation of the local structure around an edge. The time to calculate these is a huge bottleneck for this algorithm, so it would be interesting to exclude them from the feature set and observe, how relevant these counts are for the solution quality.

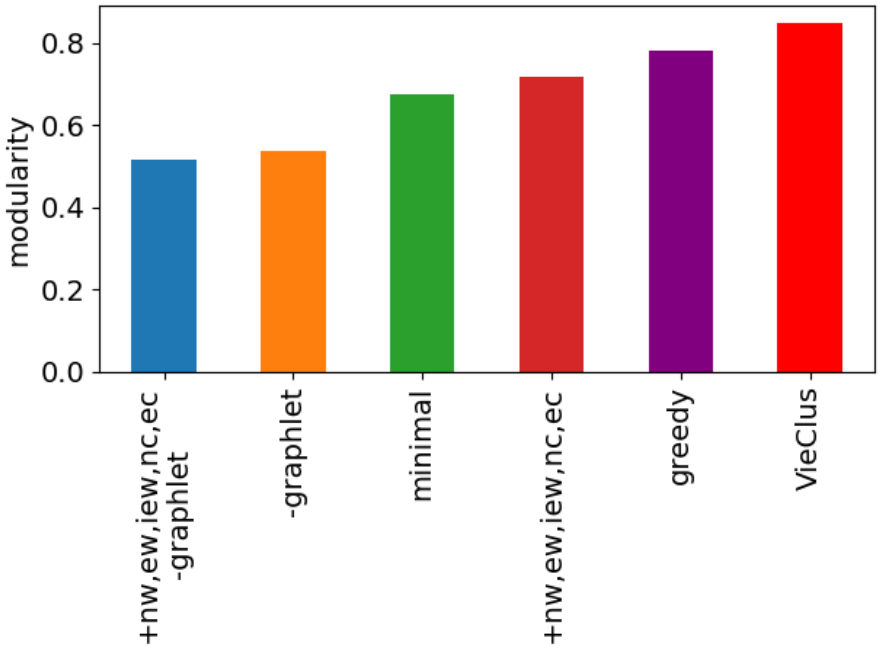


**Figure 5.9:** Two feature sets each with and without graphlet count feature mean modularities.

In Figure 5.9 we can immediately see, how big the benefit of using graphlet counts is. In the two feature sets, the achieved modularity significantly drops after removing the graphlets (...-graphlets). Additionally, the features added to the minimal feature set (nw, ew, nc, ec, iew) seem to only have a good influence if the graphlets are present. After discarding the graphlet information, the algorithm still performs better than a model with random weights. From this, we can conclude that a lot of information is gained from the graphlet counts, although some of the other features in the minimal feature set also have relevance for creating a clustering.

## 5.5 Overfitting and Regularization

When using a network without regularization it can achieve nontrivial results in the clustering process compared to a randomly initialized network in Figure 5.10. The errors of instances from the same graph behave similarly, even for instances not used in training. For data instances stemming from other graphs not included in the training set, the errors are noticeably higher.

### 5.5.1 L2 Regularization

If we add this regularization, the clustering results improve.

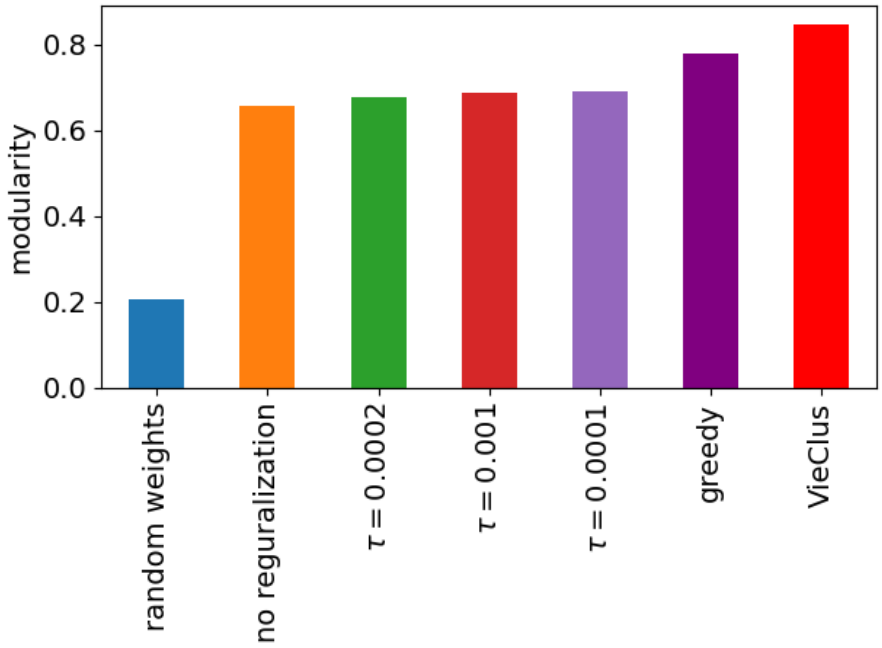


**Figure 5.10:** Geometric Mean modularities for different L2 regularization hyperparameters and comparison with random weights, minimal feature set.

In Figure 5.10 it is visible, that there is some clear benefit in using L2 regularization. All runs with regularization are consistently better than the runs without. Here it would be interesting to see, how well the algorithm can perform with a better choice of the regularization parameter $\tau$. Also, it is obvious, that the model is indeed learning important criteria from the training data, as the modularity is significantly higher than using a model with random weights.

### 5.5.2 Dropout

Although Dropout usually helps to prevent overfitting, in our setting it does not work as well as L2 Regularization. Therefore, it was not used in further experiments.

## 5.6 Switching between XGBoost and Neural Network

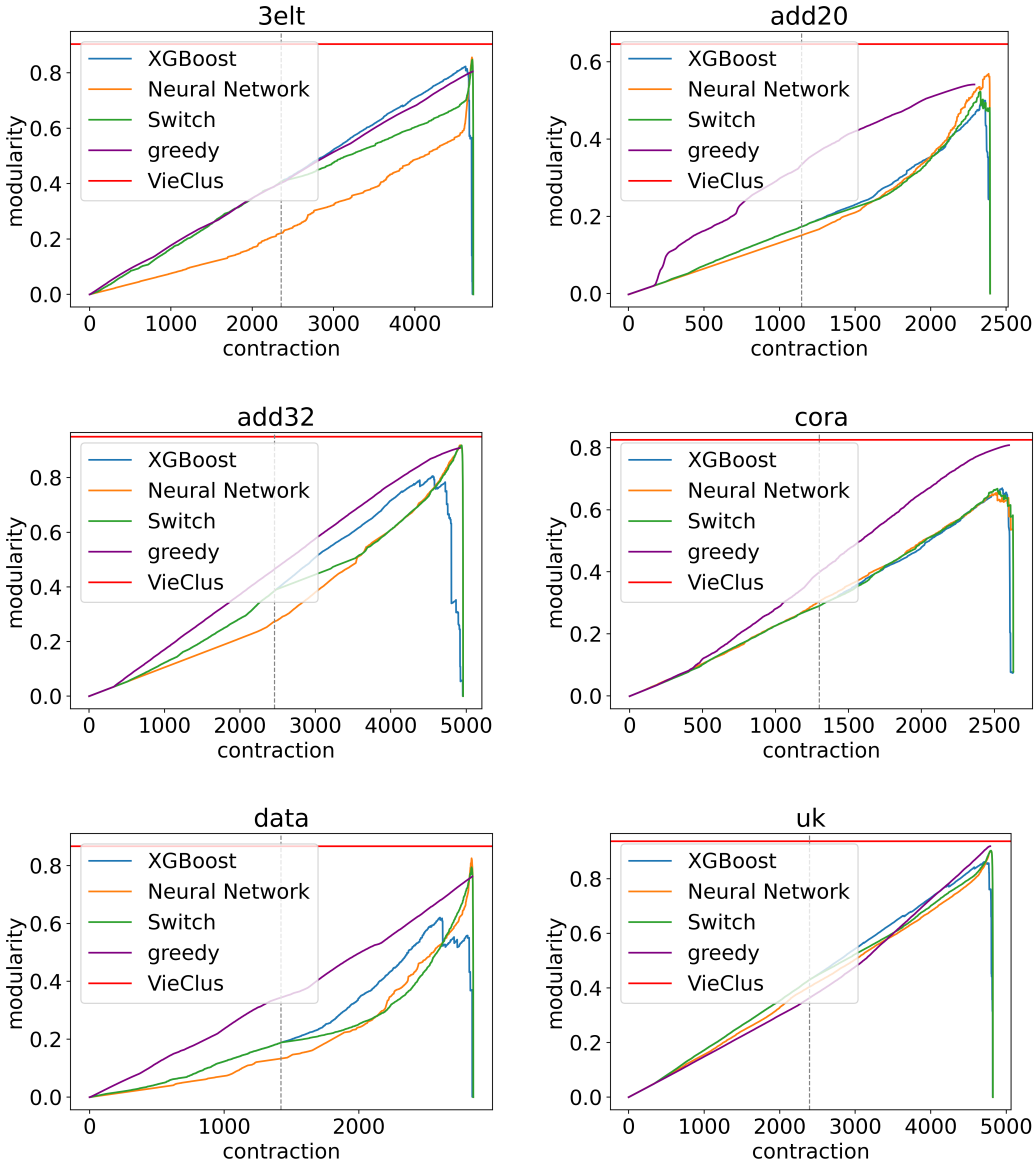


**Figure 5.11:** Modularity Development on different graphs XGBoost vs Neural Network vs Switch at 50% progress, minimal+nw,ew feature set.

As visible in Figure 5.11, the modularity for the clustering produced by XGBoost is higher than the modularity for the Neural Network initially. If the XGBoost model is better suited for the beginning phase of clustering and the neural network excels more at the end of clustering, we could produce even better results than with the models separately. As we

can see in Figure 5.11, the modularity follows the one from XGBoost up to 50%. After we switch over to the neural network, we see a drop and the result is worse for all instances except for the cora graph than just using the neural network model. This is the only graph instance, where XGBoost performs better than the neural network, so it seems like it could contract some nodes in the first half, that prepare the graph for a better solution than with just the neural network.

## 5.7 Network Size

If the network is significantly smaller, regularization is not as necessary as for our final model. This is most likely due to the fact, that the model is just not capable of overfitting and is forced to generalize. With a smaller model with fewer or smaller layers, the training error and results were not as good as for our current model. For bigger networks, the training error did not significantly change, but overfitting became more of a problem. Changing the size of the network could produce better results than we can, but due to time and hardware limitations, extensive optimization of these factors is not discussed in this thesis.

## 5.8 Replicate Clusterings

To explore how well the algorithm can adapt to a graph and clustering, we provide the model with training data of a graph and run the algorithm with this model on the same graph. For this experiment, we train the model 20 epochs instead of 2 as we have significantly less training data and thus need more epochs to achieve similar convergence. Some results are visualized in Figure 5.12. For the first two, an edge is green, if the training clustering and the replicated clustering agree on the two nodes to be in the same cluster. The edge is colored red if the clusterings disagree. The add32 graph has a tree-like structure with small branches extending out of a long path. The clusterings agree on almost all extensions as they are largely green. Our model is thus able to identify very clear structures for this graph as clusters the same way VieClus defines them and this matches with the basic intuition. In some cases, there are disagreements between the solutions, for example in the long upper branch. The main path has a lot of red edges and the two clusterings often disagree in this area so our model is not able to cluster path nodes correctly with their extending branches and other path nodes. For most graphs, the replications look similar to the second example. There are a lot of edges that disagree. Also, some parts that seem very separated from other parts are almost completely red. The lower right extension is an example of this. Although a human would probably group all these nodes together, our algorithm decides for the edges in this part of the graph to be singletons. This happens probably because the graph has a grid-like structure and it is hard for our algorithm to distinguish if a node is inside the big patch in the middle or at the end of the extension.

Another problem one encounters is visible in the 3elt graph below. Around 67% of all nodes are contained inside the blue cluster. All other nodes are singletons. This might happen because one huge cluster is created and reaches a local modularity optimum. All later contractions of singletons are discarded.

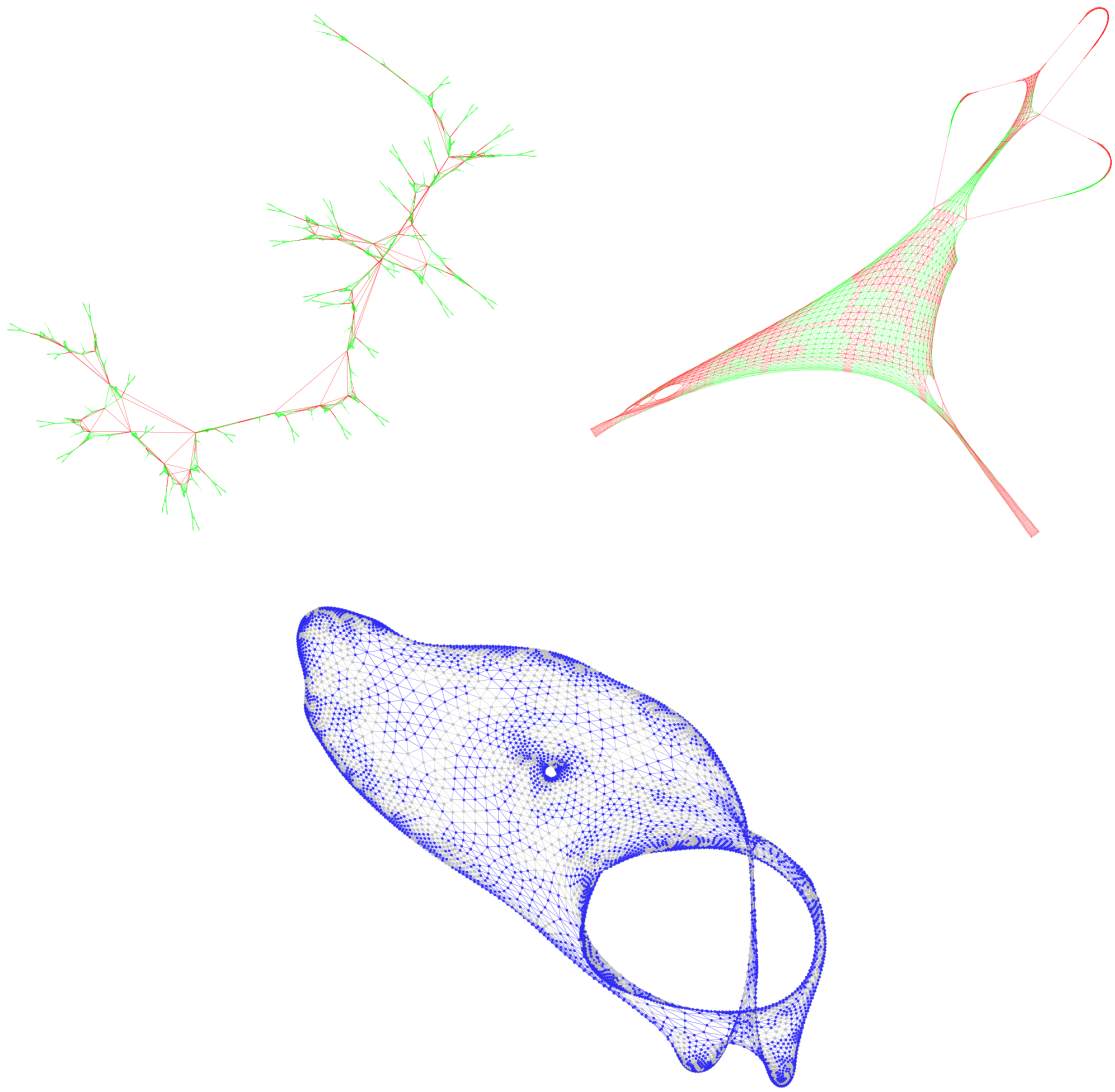

**Figure 5.12:** add32, data and 3elt after training the model on each individually and clustering them with the respective model.

| | ours | | | greedy agglomeration | | |
|---|---|---|---|---|---|---|
| Graph | t(s) | cov | mod | t(s) | cov | mod |
| Evaluation Graphs | | | | | | |
| chesapeake | 0.026 | **0.741** | 0.179 | 0.002 | 0.594 | **0.238** |
| dolphins | 0.045 | **0.881** | 0.365 | 0.003 | 0.761 | **0.512** |
| lesmis | 0.049 | 0.618 | 0.365 | 0.005 | **0.795** | **0.515** |
| adjnoun | 0.160 | 0.249 | 0.131 | 0.011 | **0.485** | **0.298** |
| football | 0.153 | 0.540 | 0.271 | 0.012 | **0.708** | **0.552** |
| jazz | 1.768 | **0.929** | 0.277 | 0.041 | 0.779 | **0.439** |
| celegansneural | 1.924 | 0.313 | 0.182 | 0.080 | **0.671** | **0.385** |
| celegans_metabolic | 3.427 | 0.363 | 0.210 | 0.143 | **0.665** | **0.412** |
| email | 24.149 | 0.321 | 0.246 | 0.859 | **0.730** | **0.515** |
| netscience | 6.173 | 0.987 | 0.938 | 0.788 | **0.989** | **0.955** |
| add20 | 87.363 | 0.771 | **0.569** | 3.249 | **0.781** | 0.541 |
| cora | 48.012 | 0.715 | 0.654 | 3.744 | **0.884** | **0.808** |
| data | 96.408 | 0.905 | **0.825** | 4.511 | **0.965** | 0.763 |
| 3elt | 169.044 | 0.914 | **0.856** | 12.626 | **0.969** | 0.805 |
| uk | 97.489 | 0.947 | 0.903 | 12.824 | **0.963** | **0.920** |
| power | 86.855 | 0.941 | 0.905 | 11.876 | **0.965** | **0.934** |
| add32 | 133.537 | 0.951 | **0.919** | 12.633 | **0.960** | 0.908 |
| hep-th | 311.714 | 0.791 | 0.745 | 28.011 | **0.906** | **0.830** |
| PGPgiantcompo | 845.056 | 0.746 | 0.719 | 66.208 | **0.926** | **0.850** |
| wing_nodal | 2,365.970 | 0.829 | **0.720** | 69.924 | **0.944** | 0.666 |
| astro-ph | 8,654.730 | 0.651 | 0.614 | 215.191 | **0.819** | **0.705** |
| cond-mat | 1,928.300 | 0.733 | 0.714 | 172.561 | **0.880** | **0.776** |
| as-22july06 | 21,827.100 | 0.656 | 0.444 | 385.898 | **0.794** | **0.636** |
| cond-mat-2003 | 12,461.800 | 0.592 | 0.572 | 748.109 | **0.862** | **0.772** |
| cond-mat-2005 | 31,013.500 | 0.534 | 0.515 | 1,338.040 | **0.845** | **0.789** |
| t60k | 16,511.800 | 0.955 | **0.935** | 2,026.410 | **0.988** | 0.921 |
| Geometric Mean | 51.191 | 0.672 | 0.489 | 3.077 | **0.820** | **0.633** |
| Feasible Training Graphs | | | | | | |
| 3-cluster | 0.020 | **0.926** | 0.392 | 0.000 | 0.926 | **0.571** |
| zachary | 0.107 | 0.449 | 0.168 | 0.001 | **0.692** | **0.392** |
| polbooks | 0.310 | 0.576 | 0.342 | 0.008 | **0.893** | **0.523** |
| whitaker3 | 680.933 | 0.934 | **0.867** | 43.386 | **0.979** | 0.811 |
| crack | 737.058 | 0.921 | **0.851** | 52.291 | **0.972** | 0.832 |
| fe_4elt2 | 837.790 | 0.940 | **0.895** | 56.201 | **0.976** | 0.809 |
| 4elt | 1,665.680 | 0.951 | **0.903** | 127.444 | **0.983** | 0.816 |
| fe_sphere | 1,942.050 | 0.870 | **0.826** | 129.050 | **0.979** | 0.812 |
| cti | 1,928.810 | 0.904 | 0.834 | 144.233 | **0.948** | **0.843** |
| memplus | 8,488.120 | 0.664 | 0.624 | 165.903 | **0.787** | **0.630** |
| cs4 | 2,799.040 | 0.872 | 0.810 | 271.705 | **0.938** | **0.854** |
| Geometric Mean | 114.042 | 0.798 | 0.614 | 4.998 | **0.911** | **0.698** |

**Table 5.3:** Time, Coverage and Modularity comparison of clustering results ours vs greedy agglomeration.

| | ours | | greedy agglomeration | |
|---|---|---|---|---|
| Graph | perf | icc | perf | icc |
| Evaluation Graphs | | | | |
| chesapeake | 0.607 | **0.600** | **0.702** | 0.472 |
| dolphins | 0.564 | **0.855** | **0.807** | 0.222 |
| lesmis | **0.887** | 0.000 | 0.752 | **0.726** |
| adjnoun | **0.884** | 0.000 | 0.833 | **0.420** |
| football | 0.750 | 0.000 | **0.891** | **0.616** |
| jazz | 0.567 | **0.400** | **0.758** | 0.162 |
| celegansneural | **0.867** | 0.000 | 0.745 | **0.408** |
| celegans_metabolic | **0.853** | 0.000 | 0.793 | **0.403** |
| email | **0.954** | 0.000 | 0.808 | **0.545** |
| netscience | 0.980 | 0.750 | **0.988** | **0.923** |
| add20 | 0.813 | 0.400 | **0.901** | **0.556** |
| cora | **0.941** | 0.000 | 0.939 | **0.769** |
| data | **0.926** | 0.824 | 0.794 | **0.899** |
| 3elt | **0.943** | 0.718 | 0.837 | **0.890** |
| uk | 0.957 | 0.890 | **0.958** | **0.912** |
| power | 0.966 | 0.880 | **0.970** | **0.905** |
| add32 | **0.969** | 0.137 | 0.954 | **0.936** |
| hep-th | **0.980** | 0.000 | 0.970 | **0.783** |
| PGPgiantcompo | **0.978** | 0.000 | 0.952 | **0.750** |
| wing_nodal | **0.892** | 0.702 | 0.726 | **0.936** |
| astro-ph | **0.912** | 0.000 | 0.889 | **0.500** |
| cond-mat | **0.991** | 0.000 | 0.949 | **0.713** |
| as-22july06 | 0.810 | 0.000 | **0.872** | **0.519** |
| cond-mat-2003 | **0.995** | 0.000 | 0.908 | **0.462** |
| cond-mat-2005 | **0.997** | 0.000 | 0.888 | **0.429** |
| t60k | **0.980** | 0.910 | 0.932 | **0.970** |
| Geometric Mean | **0.872** | 0.000 | 0.862 | **0.595** |
| Feasible Training Graphs | | | | |
| 3-cluster | 0.654 | **0.900** | **0.860** | 0.833 |
| zachary | 0.725 | 0.000 | **0.786** | **0.533** |
| polbooks | **0.797** | 0.000 | 0.750 | **0.250** |
| whitaker3 | **0.934** | **0.871** | 0.833 | 0.593 |
| crack | **0.931** | 0.767 | 0.860 | **0.921** |
| fe_4elt2 | **0.955** | 0.889 | 0.834 | **0.968** |
| 4elt | **0.952** | **0.905** | 0.834 | 0.610 |
| fe_sphere | **0.956** | 0.809 | 0.834 | **0.975** |
| cti | **0.931** | 0.829 | 0.896 | **0.904** |
| memplus | **0.970** | 0.424 | 0.962 | **0.625** |
| cs4 | **0.937** | 0.827 | 0.917 | **0.917** |
| Geometric Mean | **0.879** | 0.000 | 0.850 | **0.694** |

**Table 5.4:** Performance and inter-cluster conductance comparison of clustering results ours vs greedy agglomeration.

## 5.9 Evaluation on more Graphs

In Table 5.3 and Table 5.4 we can see the results of clustering the graph with our algorithm and with the greedy agglomeration. We see that adding the feature generation and searching for the highest score to the algorithm increases the running time drastically as we can expect. The greedy algorithm achieves in almost all instances higher coverage. For modularity, we can see that in some cases our algorithm produces a clustering with a higher score. Interestingly, in no instance with higher modularity, our algorithm has higher coverage than the greedy algorithm. So the greedy algorithm can move more edges inside of clusters, but if our algorithm works properly, it has a better structural understanding of the graph. For some instances used in training, we also see a formation of very good clusterings results regarding modularity beginning with the whitaker3 graph and ending with the fe_sphere graph. As the graphs are sorted by node count, this indicates that there are some sizes (around 10-16k nodes, see Table 5.2) for graphs, that perform very well with our algorithm and this trained model. As our dataset is biased towards medium-sized graphs, this could produce these results. In terms of performance our algorithm is better than the greedy agglomeration in 17 out of 26 graphs, sometimes significantly (e.g. lesmis ours: 0.887218, greedy: 0.752221). For the last metric, the inter-cluster conductance, our algorithm has very low scores of zero in many instances. This happens as some nodes form a singleton cluster at the point with the highest modularity. The singletons are contracted inside other clusters later on. However, our model decides for other contractions leading to lower modularity before the singletons are combined with other nodes. This change will not be contained in the final solution. In these cases of zero inter-cluster conductance, the modularity is also significantly smaller than for the greedy agglomeration. This implies that the clusters are not finished yet, but the algorithm is not able to make good decisions in the following steps in terms of modularity. A bigger training set could possibly produce a model, that overcomes these problems and can contract all singleton nodes into bigger clusters.

CHAPTER 6

# Conclusion

In this thesis, we present a supervised learning-based technique for graph clustering, that relies on structural edge properties. For this, we introduce an approach for data creation and data augmentation based on a given graph and clustering. After training a model on the created data, we use it as the decision rule instead of the modularity change in the greedy agglomeration framework. In our experiments, our final algorithm and model has higher modularity than the original greedy agglomeration in some test instances, and over half of the instances have better performance scores. For our model, we test two different types of models. A tree-based XGBoost model and a neural network. For both model types, we test multiple feature combinations and found well-performing sets. For the neural network, we evaluate the effect of removing the computational expensive graphlets from the feature set and see the importance of this feature. Further, we compare both model types against each other. Our finding is, that although the XGBoost model can create good solutions, the neural network can extract more valuable information leading to better modularity results for our training set. For the neural network, we additionally test the influence of L2 regularization and Dropout to prevent overfitting with the hope to achieve better results on unseen graphs. We notice an improvement compared to no regularization with L2 regularization but are unable to find a hyperparameter working for Dropout. We test one possibility of using multiple models in the same clustering process, as we switch from an XGBoost model to a neural network halfway through the clustering process. The XGBoost model has higher modularity in the clustering process beginning than the pure neural network, but after switching to a neural network the modularity experiences drops, and the overall achieved modularity is not as high as for the pure neural network in almost all cases. To explore how accurately the model learns a clustering for a specific graph we try to replicate some clusterings with the help of a model. For one graph the model achieves a reasonable solution that had some similarities with the original clustering, while for the other graphs the solution often greatly differs. In the end, we cluster all available feasible graphs and point out some properties of the results. Our algorithm achieves higher modularity for six evaluation instances. It has lower modularity in the geometric mean

(ours: 0.489, greedy: 0.633) as our solutions for some instances are significantly worse than the ones from greedy agglomeration. The geometric mean of the performance metric of our solutions is slightly higher than the modularity maximization (ours: 0.872, greedy: 0.862).

## Future Work

An obvious improvement would be to add more graphs to the training set. We could also increase the training set by creating more diverse coarsened versions of the graphs. Currently, we have only one branch of coarsening, but we could use e.g. random matching or other algorithms to create a huge variety of coarsened graphs. Also, it would be interesting to identify the graphlets that are significant for good results. Another target for the future can be to improve the speed of the algorithm, in order to cluster bigger instances. This could be possible by the earlier mentioned update policy for the features and different data structures for maintaining the features and edges. If the speed can be improved enough, a Reinforcement Learning approach would also be possible. The algorithm is presented with a graph and a clustering for this graph. Then the algorithm contracts nodes it currently predicts to belong together. The model is rewarded for every edge, that is contracted correctly according to the provided clustering. A punishment happens for every edge that is contracted although the clustering disagrees with this decision. With this setup, the model could learn more details about how a clustering should look like. This method has the benefit, that we do not need to provide explicit coarsened versions anymore, as the algorithm creates coarsened versions on its own in a way that fits the current state of the model.

# Bibliography


[1] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning (hierarchically clustered) complex networks via size-constrained graph clustering. *J. Heuristics*, 22(5):759–782, 2016.

[2] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *CoRR*, abs/2205.13202, 2022.

[3] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *J. Comput. Virol.*, 7(4):233–245, 2011.

[4] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *SIGKDD Explor.*, 5(1):59–68, 2003.

[5] Satu Elisa Schaeffer. Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, 2007.

[6] Petr Novák, Pavel Neumann, and Jirí Macas. Graph-based clustering and characterization of repetitive sequences in next-generation sequencing data. *BMC Bioinform.*, 11:378, 2010.

[7] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 09 2006.

[8] Felix Hausberger, Marcelo Fonseca Faraj, and Christian Schulz. A distributed multilevel memetic algorithm for signed graph clustering. *CoRR*, abs/2208.13618, 2022.

[9] Gramoz Goranci, Monika Henzinger, Dariusz Leniowski, Christian Schulz, and Alexander Svozil. Fully dynamic $k$-center clustering in low dimensional metrics. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 143–153. SIAM, 2021.

[10] 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, oct 2008.

[11] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[12] Marco Gaertler. Clustering. In Ulrik Brandes and Thomas Erlebach, editors, *Network Analysis: Methodological Foundations [outcome of a Dagstuhl seminar, 13-16 April 2004]*, volume 3418 of *Lecture Notes in Computer Science*, pages 178–215. Springer, 2004.

[13] Mark E. J. Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 69(2):026113, 2004.

[14] Stijn Marinus Van Dongen. *Graph clustering by flow simulation*. PhD thesis, Dutch National Research Institute for Mathematics and Computer Science, University of Utrecht, Netherlands, 2000.

[15] Ravi Kannan, Santosh S. Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, 2004.

[16] Ravi Kannan, Santosh S. Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, 2004.

[17] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[19] George Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control. Signals Syst.*, 5(4):455, 1992.

[20] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[21] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):066111, dec 2004.

[22] Kamal Berahmand, Sogol Haghani, Mehrdad Rostami, and Yuefeng Li. A new attributed graph clustering by using label propagation in complex networks. *J. King Saud Univ. Comput. Inf. Sci.*, 34(5):1869–1883, 2022.

[23] Daniel Delling, Robert Görke, Christian Schulz, and Dorothea Wagner. Orca reduction and contraction graph clustering. In Andrew V. Goldberg and Yunhong Zhou, editors, *Algorithmic Aspects in Information and Management, 5th International Conference, AAIM 2009, San Francisco, CA, USA, June 15-17, 2009. Proceedings*, volume 5564 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2009.

[24] Alex Pothen, Horst D Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM journal on matrix analysis and applications*, 11(3):430–452, 1990.

[25] Xin Jin and Jiawei Han. *K*-means clustering. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning and Data Mining*, pages 695–697. Springer, 2017.

[26] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442, 1998.

[27] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. Efficient graphlet counting for large networks. In Charu C. Aggarwal, Zhi-Hua Zhou, Alexander Tuzhilin, Hui Xiong, and Xindong Wu, editors, *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*, pages 1–10. IEEE Computer Society, 2015.

[28] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

[29] Chris Walshaw and Mark Cross. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000.

[30] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In Reda Alhajj and Jon G. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018.

[31] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Mag.*, 29(3):93–106, 2008.

[32] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2011.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[34] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress, 2010.

[35] Sonja Biedermann, Monika Henzinger, Christian Schulz, and Bernhard Schuster. Memetic graph clustering. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L'Aquila, Italy*, volume 103 of *LIPIcs*, pages 3:1–3:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

# Appendices

APPENDIX A

# Implementation Details

## CUDA support

The training and clustering were performed on a device that supports CUDA. Although the code can theoretically run on a CPU, the running time is increased drastically, and thus usage of a CUDA device is the default for our code.

## File Formats

The implementation assumes all graphs to be in the Metis graph format that is also used by VieClus [35] and KaHIP [28]. The graph has to be undirected, without self-loops and no parallel edges connecting two nodes multiple times. Although the algorithm could work with weighted nodes and edges it is currently not supported. Therefore any weights should be removed beforehand.

## Score Tie Resolution

In case more than one edge receives the maximum score of an iteration, an edge is uniform randomly picked among the edges with the maximum score to resolve the tie. With this, another random seed can produce a different clustering solution.

## Implementation Data Structure

The basic algorithm works without recalculating the modularity change or sorting by modularity. In our implementation, we use heaps similar to the original greedy agglomeration to keep track of possible modularity changes.

APPENDIX B

# Further Results

## Clustering Visualization

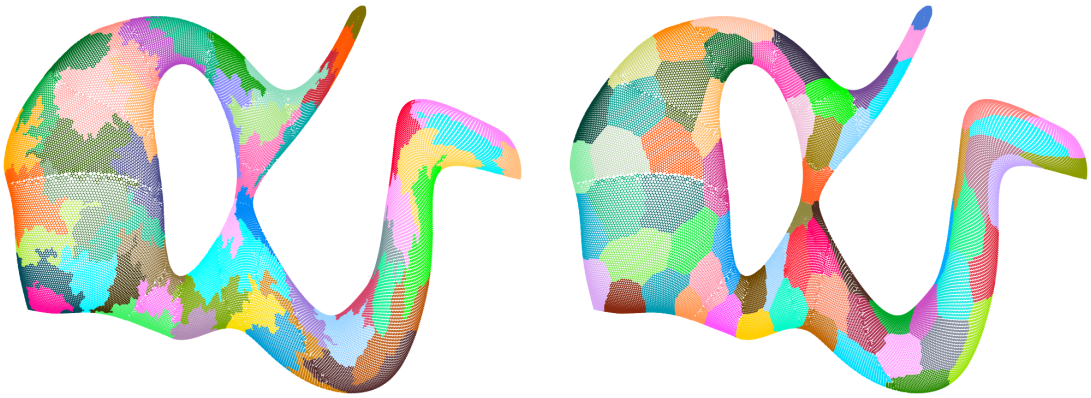

**Figure B.1:** The t60k graph clustered with our algorithm and VieClus.

We compare the clusters in the t60k graph in Figure B.1, in which our algorithm performs better than greedy agglomeration with the solution provided by VieClus. The clusters from VieClus have a clearer structure and are on average smaller (937.5 nodes) than our solution (1132.2 nodes). Our solution does not structure the graph into strict blocks of nodes. Instead, the borders of clusters often extend into other clusters. This also stretches clusters over longer areas of the graph. Another interesting observation for our algorithm is in the lower left corner. We can see a pink cluster completely contained inside of a

dark cluster. On the first view, our solution looks more organic, but as most of the graph is similar to a grid, the organic structures cannot exist due to the graph's structure but originate from our algorithm. These structures also explain the lower coverage. A thin branch of a cluster adds only a small fraction of edges to the intra-cluster edges. At the same time, it separates nodes from each other and reduces the number of edges inside a cluster.

## Only Modularity Gain

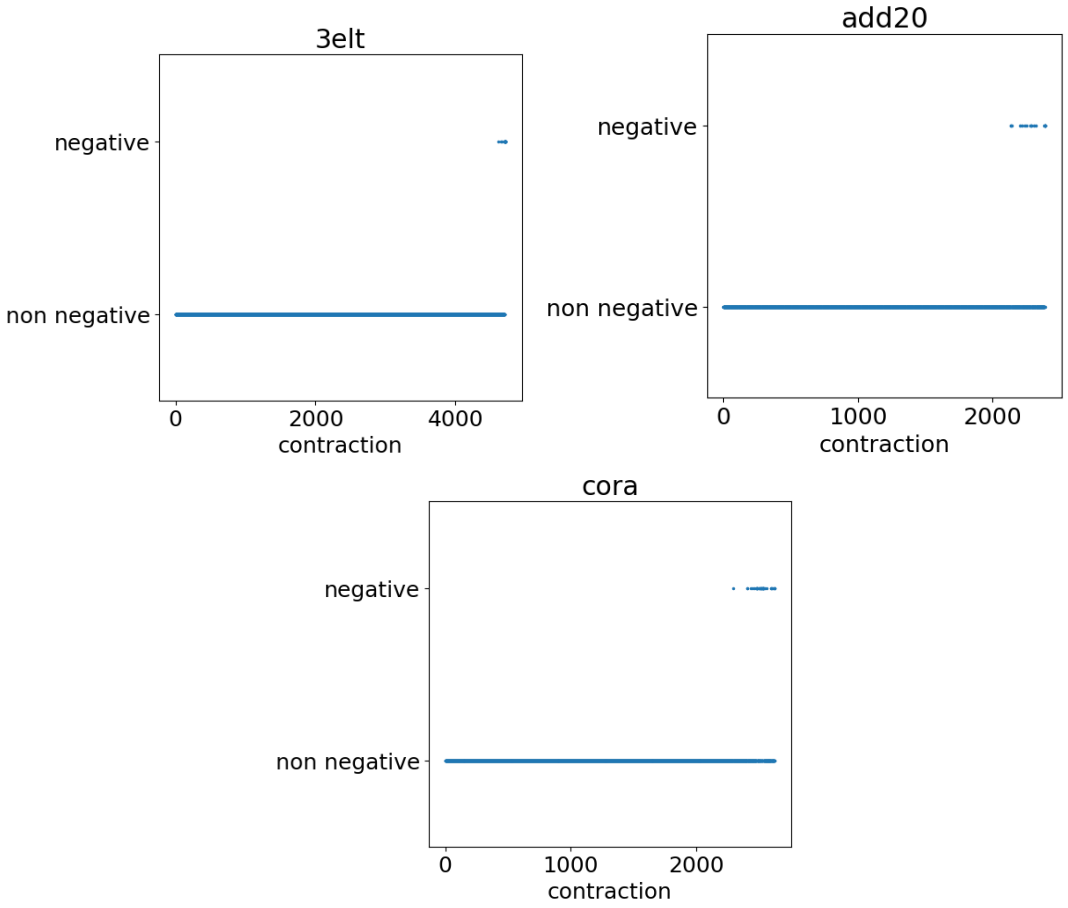


**Figure B.2:** Appearance of negative modularity gain in the clustering process.

Small drops followed by good decisions increasing the modularity again lead to plateaus in the modularity. The real potential of the method might not be reached due to insufficient training data or hyperparameters. In Figure B.2, all of the contractions lead to a non-negative change for most of the process. The drops in modularity are only appearing in the final phase of the clustering process. To prevent bad decisions close to the end, we restrict the model to only contract edges, that yield no modularity drop. As most of the

contractions already follow this rule, we also do not change the initial part of the process but only the final phase.

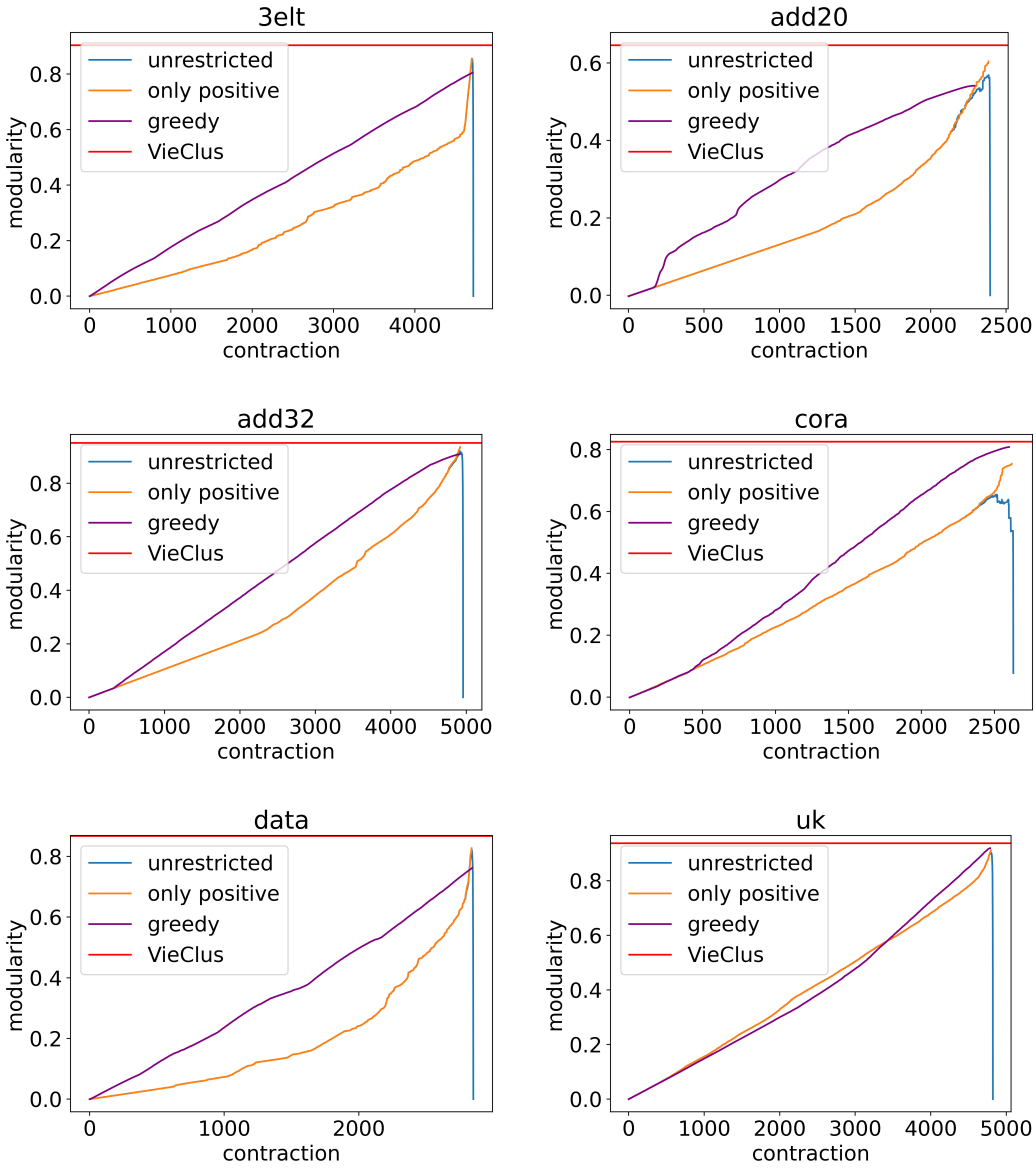


**Figure B.3:** Modularity Development on different graphs with and without positive modularity restriction (neural network, minimal+nw,ew feature set).

In the beginning, the modularity behaves the same for both versions as we expect in Figure B.3. There is a huge improvement in both the cora and add20 graph instance. The unrestricted version has some plateaus for these graphs. The restricted version separates at these unsteady parts and rises significantly higher. For the graphs with no plateau, the

modularity does not increase significantly. We use the modularity metric for filtering, the model's score still makes the final decision. It seems as if the model could especially benefit from more training data of coarse graphs, as only at the coarse level do the negative-gain decisions start.