TRƯỜNG ĐẠI HỌC FPT

Capstone Project Report

An Improvement of Cluster-GCN with constraints

Students Name

Nguyen Bao Phuoc - HE153036
Duong Thuy Trang - HE150573
Nguyen Thanh Tung - HE150163

Under the supervision of

Associate Professor Dr. Phan Duy Hung

Bachelor of Artificial Intelligence
FPT University - Hoa Lac Campus
Spring 2023

DECLARATION

Project Title	An Improvement of Cluster-GCN with constraints
Author (Student ID)	*Nguyen Bao Phuoc (HE153036), Duong Thuy Trang (HE150573), Nguyen Thanh Tung (HE150163)*
Supervisor	Associate Professor Dr. Phan Duy Hung

We declare that this thesis entitled *An Improvement of Cluster-GCN with constraints* is the result of our own work except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Nguyen Bao Phuoc, Duong Thuy Trang,
Nguyen Thanh Tung

Department of Artificial Intelligence
FPT University - Hoa Lac Campus

Date: April 26, 2023

ACKNOWLEDGEMENTS

We would like to thank our supervisor *Assoc. Prof. Hung* for his consistent support and guidance during the making of this thesis as well as the time he taught us foundation courses in Machine Learning. We are also thankful to all staff of FPT University for working hard to make our four years of college memorable. We have grown a lot from inexperienced freshmen who barely knew the world outside our circles. Last but not least, we cannot express our heartfelt gratitude enough to families, friends, brothers and sisters who silently make our lives more comfortable, laughed with us, cried with us, spend the whole night disentangling the complexities of adulthood but never hinder our desires to live the life we wanted. The bad news is, at the moment this thesis is accepted, we are now alone. But do you know what is more exciting, we are now alone! Those sweet and youthful days are meant to be left behind as everyone is about to embark on their adventure. To conclude, we cannot choose a word carefully enough to thank all the unconditional support in this intense academic year.

Nguyen Bao Phuoc, Duong Thuy Trang, Nguyen Thanh Tung

Hoa Lac Campus – FPT University

Date: April 26, 2023

ABSTRACT

Cluster-GCN is one of the effective methods in studying the scalability of Graph Neural Networks. The idea of this approach is to use METIS community detection algorithm to split the graph into several sub-graphs that are small enough to fit into a common GPU. However, METIS algorithm still has some limitations. Therefore, this project proposes Leiden algorithm as an alternative, which was scientifically published 21 years after METIS's and claimed to be powerful in identifying communities in networks. However, the common feature of community detection algorithms makes nodes in the same community tend to be similar. For that reason, this project also proposes to add constraints such as minimum/maximum community size and overlapping communities to increase community diversity, thereby improving performance of Cluster-GCN by 0.98% ROC-AUC score on a single 8GB GPU device.

Keywords: Graph Convolution Network, Graph clustering, Leiden algorithm, Graph mining, Constrained clustering

Contents

List of Figures

List of Tables

List of abbreviations

Abbreviation	Definition
ROC	Receiver Operating Characteristic
AUC	Area Under the Curve
GNNs	Graph neural networks
GCNs	Graph convolutional networks
SMP	Stochastic Multiple Partition
PPI	Protein-Protein Interactions dataset
$Leiden(.)$	A set of community extracted by the Leiden algorithm
$ER(C,\ C')$	Edge ratio between 2 community: C and C'. The definition is defined in equation 3.15
$S(\boldsymbol{\mathcal{G}})$	A set of species in graph $\boldsymbol{\mathcal{G}}$
r-Cluster-GCN	Cluster-GCN architecture with random partition
LeidenGCN	Cluster-GCN architecture with the Leiden algorithm
b-LeidenGCN	Bounded LeidenGCN. The LeidenGCN architecture with minimum/maximum community size constraints
ob-LeidenGCN	Overlapping and Bounded LeidenGCN. The b-LeidenGCN with overlapping community constraints

Chapter 1 Introduction

1.1. Problem & Motivation

1.1.1. Graph Convolutional Network approach

Graph Convolutional Networks (GCNs) [1] are receiving interest from the scientific community, as illustrated by their widespread graph-based applications in a variety of type domains, such as node classification, link prediction, and recommender system [2]. The advantage of GCNs is their ability to simultaneously capture node/edge representations and graph relational structure. Also, the layer-wise linear architecture of GCNs allows the model to learn richer and more powerful node/edge representations by stacking multiple GCN layers. Researchers have shown that adding depth to GCNs results in state-of-the-art performance [3, 4, 5]. Still, adding more layers proportionally increases runtime and memory usage. This is considered to be one of the major limitations of the GCNs architecture - the trade-off between the number of layers and computational resources. Furthermore, handling large-scale graphs by advanced Graph Convolutional Networks (GCNs) architecture is a challenging task since prediction on each node is processed with regard to information from many other nodes [2, 3, 6, 7]. Thus, effective training these models at scale requires sophisticated algorithms that are well beyond standard SGD.

1.1.2. GCN for large datasets

The original GCNs architecture operates with a full batch training method, which may require a large storage and sometimes can lead to memory overflow. To overcome this burden, several GCNs architectures based on node-wise and layer-wise sampling have been proposed [3, 6, 7]. Despite observed improvements, these methods are still affected by neighborhood expansion problem and the demanding memory for deeper networks. Notably, Cluster-GCN [2] is proposed to tackle two aforementioned issues with another approach. To be more specific, before passing the entire graph to the device for training, a clustering algorithm/community detection (e.g. METIS [8]) is used to split a graph into several clusters, then, each cluster forms a subgraph before fitting into processing units for training. By using this strategy, the amount of information used to train in each iteration is much smaller than the entire graph, which not only leads to a significant reduction in runtime and memory but also achieving outstanding performance on several large datasets such as Reddit and PPI [2]. Furthermore, those findings have inspired more deep GCN-based architecture designs to handle large datasets, take DeeperGCN [4], DCBGCN [5], RevGNN-Deep [9] and HC-GCN [10] as examples.

1.1.3. Motivation

Looking at the number of citations since its publication in 2019, the Cluster-GCN architecture [2] is undoubtedly a notable variant of GCN. This architecture has laid the foundation for advanced methods on Graph Learning [4, 5, 9, 10]. In addition to that, a good community detection algorithm such as METIS [8], Louvain [11], Leiden [12] can boost the performance of the model due to less between-partition links removal [2]. This motivates us to deep dive into graph community detection algorithms to see whether changing clustering algorithms can help the model to improve further or not. Furthermore, community detection in general leads to communities with nodes with similar characteristics. Consequently, when applied to Cluster-GCN, it leads to skew node distribution. It thus encourages us to study on adding constraints such as overlapping communities in order to balance these distributions with the belief that it may increase the convergence speed or even increase

performance of the model.

1.2. Related works

1.2.1. Graph Neural Networks (GNNs)

Graph Neural Network models have been studied since 2014; however, their drawback stems from the specific approach based on Spectral Graph Neural Network [13, 14]. Originally, GNNs utilize eigen-decomposition, eigenvector, and eigenvalues, which result in high computational cost proportional to the size of the graphs. To overcome this problem, Kipf & Welling bridged the gap between "spectral" and "spatial"" GNNs and proposed GCN architecture [1]. with the idea to extract node embeddings based on their neighbors. Since then, GCN has inspired many researchers and opened up applications in a variety of domains. Notably, some adaptations of GCN reach the pinnacle of different benchmark dataset [2, 3, 4, 6].

Graph Convolutional Networks (GCNs).

Graph Convolutional Networks are considered as one of the basic Graph Neural Network variants. This section provides understanding of GCN's mechanisms developed by Kipf & Welling.

Definition. Most GCNs variants share the general architecture. Given a graph $\boldsymbol{\mathcal{G}} = \{\boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}}\}$ which is represented by:

- A $N \times D$ node features matrix $\mathbf{X}$, where $N = |\boldsymbol{\mathcal{V}}|$ is the number of nodes in graph $\boldsymbol{\mathcal{G}}$, D is feature dimension. Each row x_i is a D dimensions array represent the feature of node i.
- An adjacency matrix **A,** which is a graph structure description in matrix form, or or an edge index sparse matrix with values indicating non-zero value in the adjacency matrix

The objective of GCN is to learn the representation of the individual node through its neighbors. Therefore, the dimension of the output matrix $\mathbf{Z}$ is corresponding to the number of nodes and embedding size, which is a $N \times E$ matrix. These embeddings represent nodes in the graph and can be further processed to deal with a specific problem such as node, edge, and graph properties prediction.

Propagation. In general, a deep Neural Network layer can be represented as a nonlinear function:

$$\mathbf{H}^{(l+1)} = f\left(\mathbf{H}^{(l)}, \mathbf{A}\right) \tag{1.1}$$

In GCNs, $\mathbf{H}^{(0)} = \mathbf{X}$ and $\mathbf{H}^{(L)} = \mathbf{Z}$, where L is the number of layers. With the idea of learning node features through their neighbors, the general propagate function of GCNs can be written as:

$$f\left(\mathbf{H}^{(l)}, \mathbf{A}\right) = \sigma\left(\mathbf{A}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \tag{1.2}$$

where $\sigma(.)$ denotes a nonlinear activation function and $\mathbf{W}^{(l)}$ is the trainable weight matrix of the l-th layer.

However, there are two limitations of this simple version of GCNs:

- Firstly, when multiplicated with **A**, the embedding of each node's entire neighbors is summed up together but not the embedding of that node. Kipf & Welling [1] address this by

adding self-connection for every node in the graph:

$$\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_n \tag{1.3}$$

where $\widetilde{\mathbf{A}}$ is the adjacency matrix of the graph $\mathcal{G}$ with added self-connections and $\mathbf{I}_n$ is the identity matrix.

- Secondly, by summing up neighbors embedding, the node feature of each node is then scaled by the number of its neighbors. In other words, the node features are not normalized, which affects convergence speed and performance of GCNs. Dealing with this problem, Kipf & Welling also apply symmetric normalization as follows:

$$\mathbf{A}' = \widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}} \tag{1.4}$$

where $\widetilde{\mathbf{D}}$ is the degree matrix that $\widetilde{\mathbf{D}}_{ii} = \sum_j \widetilde{\mathbf{A}}_{ij}$.

Considering equation (1.1), (1.2), (1.3) and (1.4) together, the authors [1] introduce propagation rule for GCNs architecture:

$$\mathbf{H}^{(l+1)} = f\left(\mathbf{H}^{(l)}, \mathbf{A}\right) = \sigma\left(\mathbf{A}'\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \tag{1.5}$$

Layer-wise linear model. A deep Neural Network based on GCNs can be built by stacking multiple layers in the form of equation (1.5). For instance, Figure 1 gives an example of a 1-layer GCNs: after the propagation is performed, each node contains information of itself and information of its neighbor. If another layer GCNs is added, each node holding the information of its neighbors is aggregated with its neighboring nodes - which also contain their neighbors information. Understandably, after **n** layers of GCN, each node contains the information of 1-hop to **n**-hop neighborhoods. Nonetheless, for most of the real-world dataset, the entire graph can be extracted with less than 8 layers GCN, which makes the aggregation less meaningful [15]. To clarify the effect of the number of layers on GCNs architecture, Kipf & Welling [1] perform some experiments on the number of GCNs and the best results are obtained with 2-3 layers, the model tends to be worse when the number of layers is more than 8 layers.

Limitation of GCNs. Considering the scale of real-world data, GCNs can suffer from slow convergence or even the out-of-memory issue [1, 2, 3]. One of the factors leading to the memory overflow of GCNs is the full-batch gradient descent, which can observe a sharp growth in memory consumption during the training process [1]. Another reason for the same problem is the layer-wise linear model architecture. Understandably, with L layers, GCNs need to store all L-hop neighbors information for each node separately, causing exponential complexity growth corresponding to the number of layers [1, 2, 3].

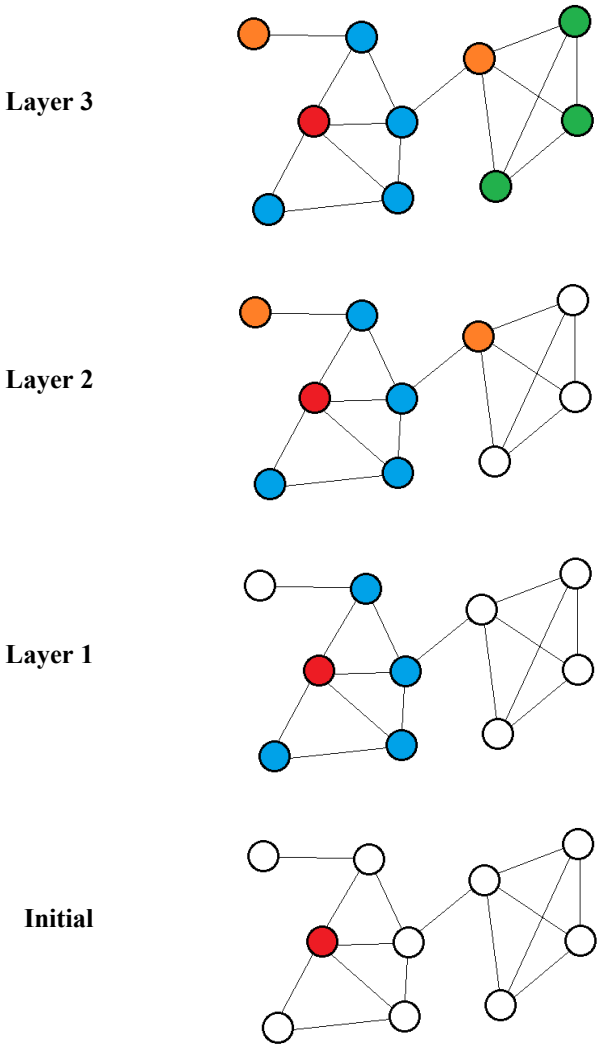


Figure 1: Propagation of GCN. Red node is the starting node, blue node indicates 1-hop neighborhood, orange is 2-hop neighborhood and green is 3-hop neighborhood.

GCN with node-wise and layer-wise sampling methodology.

Sampling in GCN. One of the intuitive ways to solve the memory issues in GCNs is sampling. Instead of using the whole graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, some sampling method can be applied get a smaller graph $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}'\}$ (where $\mathcal{G}' \subseteq \mathcal{G}$, $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$), which is small enough to fit into memory for training. Moreover, with this strategy, GCN can be trained using mini-batch gradient descent instead of full-batch, which is one of the reasons that lead to out-of-memory issues in GCNs. The above strengths of sampling have attracted a great deal of attention from researchers in the study of optimal sampling methods for GCNs architectures [2, 3, 6, 7].

Node-wise sampling architecture. One major problem of full-batch gradient descent in GCNs is the expansion of neighbors information layer by layer. Therefore, GraphSAGE [3] proposed to use mini-batch gradient descent. At each training iteration, only a subset $\mathcal{V}' \subseteq \mathcal{V}$ is used for computed propagation, which significantly reduces the size of graphs. However, the number of sampled nodes grows exponentially if all the neighbors are sampled at each layer, leading to out-of-memory issues.

To tackle these problems, GraphSAGE used a fixed-size neighbor sampling strategy. Specifically, instead of using the entire neighbor set, at each layer, a fixed-size of neighbors are sampled for computing propagation. With these strategy, the memory complexity is reduced with scale from $|\boldsymbol{\mathcal{N}}(v)|^L$ to k^L where k is the size of sampled neighbor and $\boldsymbol{\mathcal{N}}(v)$ is the set of node v's neighborhood. However, the neighbor's size is reduced in a random manner that leads to bias sampling and increasing in variance [7]. Therefore, to further improve node-wise sampling from [3], VR-GCN [7] proposed Control variance base estimator, which maintains historical embeddings $\overline{\mathbf{h}}_v^{(l)}$ as an affordable approximation to reduce variance during sampling. Specifically, each time embedding of node v, $\mathbf{h}_v^{(l)}$ is computed, the historical embedding $\overline{\mathbf{h}}_v^{(l)}$ is updated and expect to be the same as $\mathbf{h}_v^{(l)}$ if the model's weights do not change too fast. The propagation of VR-GCN can be defined as:

$$\mathbf{H}^{(l+1)} = \sigma\left(\mathbf{A}'^{(l)}\left(\mathbf{H}^{(l)} - \overline{\mathbf{H}}^{(l)}\right) + \mathbf{A}\mathbf{H}^{(l)}\right)\mathbf{W}^{(l)} \tag{1.6}$$

where $\mathbf{A}'^{(l)}$ is the l-th layer sampled symmetric normalized adjacency matrix, $\mathbf{H}^{(l)}$ is the the node embedding at the l-th layer and $\overline{\mathbf{H}}^{(l)} = \left\{\overline{\mathbf{h}}_1^{(l)}, \dots, \overline{\mathbf{h}}_N^{(l)}\right\}$ is the stack of the historical embeddings. With Control variance base estimator, VR-GCN can successfully reduce neighborhood's sampling size to $k = 2$ while keeping acceptable performance [2].

Layer-wise sampling architecture. Another sampling method that is also noticed by researchers is layer-wise sampling, with the representation of FastGCN [6]. In this architecture, the authors address the out-of-memory training issues by assuming the input graph is an induced subgraph made of vertices that are i.i.d. sampled from a possibly infinite graph under some probability [16]. Specifically, FastGCN introduces probability measures for each node in the graph, which turn graph convolution perspective into integral transform (Figure 2). With this importance sampling strategy, FastGCN can perform sampling for each layer independently while keeping the probability the same. Furthermore, [6] state that the implementation of stochastic gradient descent is according to the additivity of the loss function for independent data samples, therefore, i.i.d samples will help the model fit better with this training method.

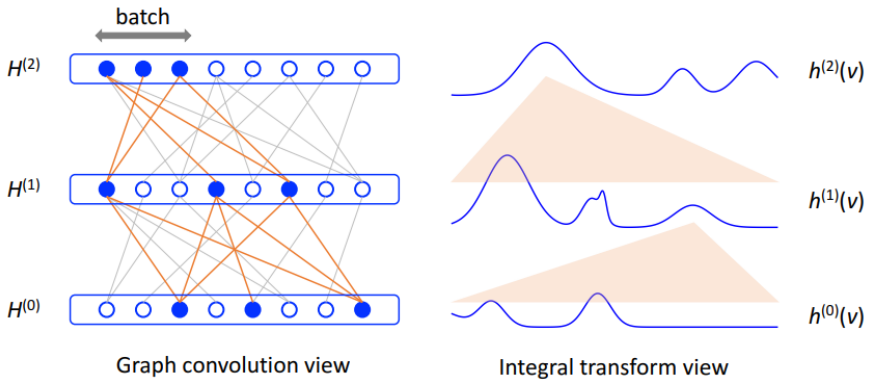


Figure 2: Two views of GCN in FastGCN. Blue nodes indicate sampled nodes, orange line is the sampled edges. In the integral transform view, embedding of the next layer is the integral transform of the previous layer (represented by an orange triangle). Picture from paper [6].

Limitation. With node-wise sampling strategy in GraphSAGE [3] and the improvement of layer-wise importance sampling in FastGCN [6], the memory requirements are significantly reduced. However, these strategies still suffer from the neighborhood expansion problem when GCNs go deep. To be more clear, consider a L layers GCNs architecture, to compute loss for a single node *v*, it required all *v*'s neighbors embedding from layer L-1, which again need all neighbors embedding from layer L-2, .etc. Even with the improvements from FastGCN [6] and the small sampling size, the overhead of these strategies is still large and time complexity will be increased exponentially corresponding to the number of layers. Furthermore, despite FastGCN [6] solving the neighborhood expansion problem better than GraphSAGE [3] (by using fixed-size layer sampling), its independent layer sampling leads to the deletion of correlation between layers. Contrary to GraphSAGE [3] and FastGCN [6], VR-GCN [7] succeeded in reducing sampling size to 2 while keeping the acceptable performance, therefore, the time complexity of this architecture is smaller than these methods above. However, VR-GCN [7] is required to store all hidden embeddings of all nodes in memory, which leads to bad memory usage (linearly increasing corresponding to the size of input graph).

1.2.2. Community Detection

Community detection is an important research problem that spans many areas, and it has been studied extensively over the last few years. The aim of community detection algorithms is to identify the modules and, possibly, their hierarchical organization, in a graph. In 2004, Girvan and Newman proposed the modularity metric [17], which is one of the most used and the best known functions to quantify community structure in a graph.

Why Community Detection?

Community detection can be used to detect groups with similar properties and extract groups based on preference, which may bring several benefits when analyzing a network. For instance, in protein-protein interaction network, the discovery of commonly interactive groups of protein provide insights for researchers in designing target drugs. Moreover, community detection algorithms can be used as the sampling phase for Cluster-GCN [2] in order to improve the performance of the model on large-scale datasets.

Definition.

A community is a subset of nodes from a graph that are densely connected in a knit group and loosely connected to others. In graph theory, a graph $\boldsymbol{\mathcal{G}} = \{\boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}}\}$ is supposed to have a community structure if it is able to divide nodes into communities. Specifically, $\boldsymbol{\mathcal{G}}$ can naturally divided into multiple subgraphs:

$$\boldsymbol{\mathcal{G}}_i = \{\boldsymbol{\mathcal{V}}_i, \boldsymbol{\mathcal{E}}_i \mid \boldsymbol{\mathcal{V}}_i \subseteq \boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}}_i \subseteq \boldsymbol{\mathcal{E}}, |\boldsymbol{\mathcal{E}}_i| \gg |E(\boldsymbol{\mathcal{V}}_i, \boldsymbol{\mathcal{V}} - \boldsymbol{\mathcal{V}}_i)|\}$$

where $|\boldsymbol{\mathcal{E}}_i|$ is the number of edge in subgraph $\boldsymbol{\mathcal{G}}_i$, $E(\boldsymbol{\mathcal{V}}_i, \boldsymbol{\mathcal{V}} - \boldsymbol{\mathcal{V}}_i)$ is the set of edges between nodes from subgraph $\boldsymbol{\mathcal{G}}_i$ and nodes outside community i in the same graph. The objective of community detection is to find these communities within the graphs.

Community Detection versus Clustering.

There are arguments that Community Detection and Clustering are similar; however, the two methods are distinguishable. On the one hand, clustering is a machine learning technique used to group similar data points into the same cluster based on their characteristics. On that basis, clustering can be applied to networks where data points are nodes and attributes are the relationship between them; beside, it is a broader field in unsupervised machine learning which deals with multiple attribute

types. On the other hand, community detection is designed specially for network analysis, taking into account a single attribute called edges or links between nodes. Nevertheless, depending on the domain and problem, different pros and cons of clustering and community detection may arise.

METIS algorithm.

Intuitively, community detection's goal can be understood as dividing a graph into multiple subgraphs while minimizing the number of edges across subgraphs. Based on this criteria, Karypis and Kumar have proposed METIS algorithm [8]. METIS is an edge-cut based multilevel graph bisection algorithm, which includes three phases (Figure 3):

- Coarsening phase
- Partitioning phase
- Uncoarsening phase

Each phase of this algorithm is better explained below.

Coarsening phase. The objective of this phases is to transform original graph $\boldsymbol{\mathcal{G}} = \{\boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}}\}$ into a sequence of smaller graph $\boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_m$ where $|\boldsymbol{\mathcal{V}}_1| > |\boldsymbol{\mathcal{V}}_2| > \ldots > |\boldsymbol{\mathcal{V}}_m|$. Specifically, a set of nodes in graph $\boldsymbol{\mathcal{G}}_i$ is aggregated together to form a single node of coarser graph $\boldsymbol{\mathcal{G}}_{i+1}$. METIS defines a set of edges no two of which are incident on the same vertex as a matching and based on this to propose 4 approaches to aggregated nodes for coarser graphs: Random matching (RM); Heavy edge matching (HEM); Light edge matching (LEM) and Heavy clique matching (HCM). Among those approaches, HEM results in good initial partitions and requires the smallest overall runtime. The idea of HEM approach is reducing edge-weight of the matching by selecting edges which have a large weight. By reducing the edge-weight of the coarser graph, the number of edge-cut is also reduced. In other words, HEM aims to find the maximal matching by matching node *u* with node *v* such that the weight of the edge between *u* and *v* is maximum over all valid incident edges (heavier edge). However, this algorithm does not guarantee that the matching obtained has maximum weight [8].

Partitioning phase. After the Coarsening phase, $\boldsymbol{\mathcal{G}}_m$ is used to compute the bisection $\boldsymbol{\mathcal{P}}_m$ such that the number of edge-cut is small and each part contains approximately half of the number of edge weights. The graph $\boldsymbol{\mathcal{G}}_m$ is much smaller than original graph, however, by aggregated edge weight from finer graph to form a *multinode* in the coarser graph, $\boldsymbol{\mathcal{G}}_m$ contain sufficient information to maintain the partition equilibrium. Similar to the Coarsening phase, Karypis and Kumar [8] examine various algorithms for partitioning phases. One of these is Greedy graph growing partitioning algorithm (GGGP), which consistently finds smaller edge-cuts than the other algorithms and requires a smaller runtime. In order to divide $\boldsymbol{\mathcal{G}}_m$ into two parts of approximately equal number of edge weights, GGGP starts from a node and grows a region around it in a breadth-first manner. The process is repeated until half of the edge's weight is traveled. However, traveling in a breadth-first manner is very sensitive to the choice of initial node. Therefore, to reduce the sensitivity, instead of growing in a breadth-first way, GGGP computes the number of edge-cut reductions when a node is inserted into the growing partition and the insert order is from largest to smallest edge-cut reduce. Although the sensitivity to the initiating node is reduced compared to breadth-first manner, GGGP is still sensitive to the initiating node.

Uncoarsening phase. In this phase, the bisection $\boldsymbol{\mathcal{P}}_m$ is projected back to original graph by going backward in the sequence $\boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_m$ from the coarsening phase. Because every node in coarser graph $\boldsymbol{\mathcal{G}}_{i+1}$is a subset of nodes in finer graph $\boldsymbol{\mathcal{G}}_i$, the partition $\boldsymbol{\mathcal{P}}_i$ can be obtained through $\boldsymbol{\mathcal{P}}_{i+1}$ by

assigning every node in those subset to corresponding *multinode*'s partition in $\boldsymbol{\mathcal{P}}_{i+1}$. Specifically, assuming $\boldsymbol{\mathcal{V}}_i^v$ is a set of node in $\boldsymbol{\mathcal{G}}_i$ that aggregated together to form a *multinode* v in $\boldsymbol{\mathcal{G}}_{i+1}$:

$$\boldsymbol{\mathcal{P}}_i[u] = \boldsymbol{\mathcal{P}}_{i+1}[v] \ \forall u \in \boldsymbol{\mathcal{V}}_i^v \tag{1.7}$$

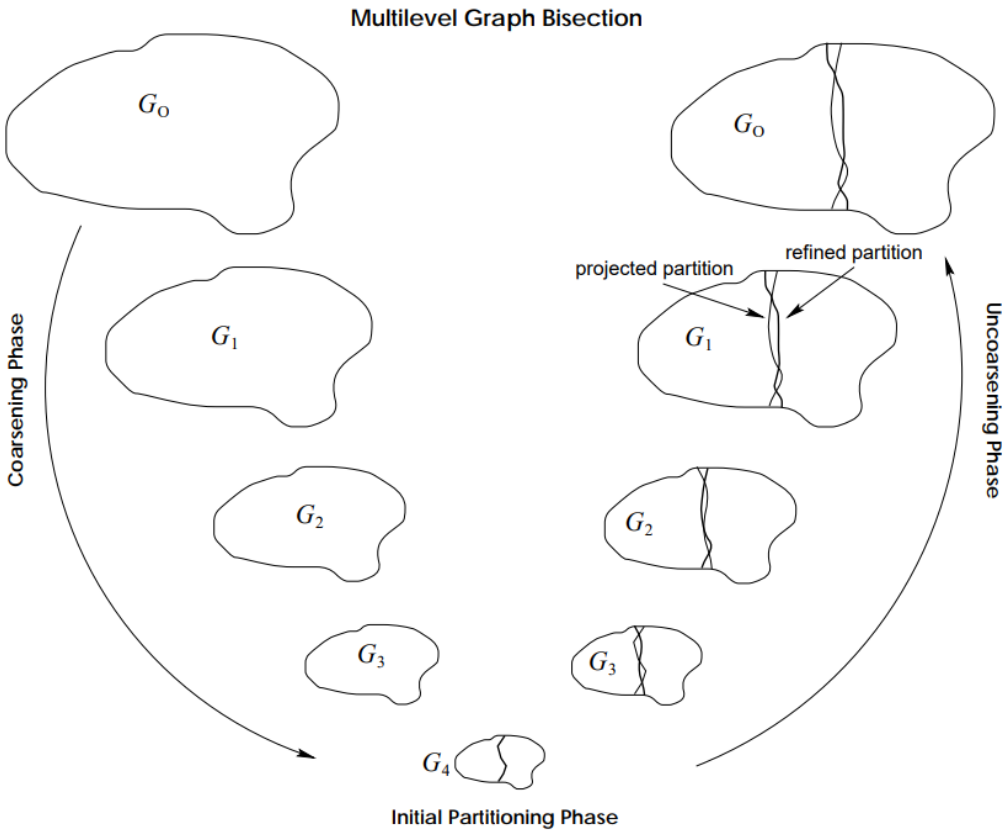


Figure 3: Illustration of METIS community detection. Picture from [8]

However, $\boldsymbol{\mathcal{P}}_i$ may not be local minimum corresponding to graph $\boldsymbol{\mathcal{G}}_i$. Therefore, a partition refinement algorithm is used after $\boldsymbol{\mathcal{P}}_i$ is obtained. The idea of this algorithm is swapping 2 nodes in different partitions if the number of edge-cut is reduced.

Limitations. Based on efficiency and fast execution time, METIS is used as a preprocessing stage in Cluster-GCN [2] with the purpose of dividing a large graph into a set of smaller subgraphs, thereby, minimizing the amount of memory requirements and improve scalability for GCN on large dataset, detail in section 3.1. However, METIS still has limitations when applied to supervised learning algorithms like GCN [1]. Such limitations may include: (1) Nodes with similar characteristics are often in the same community, so this will skew the distribution of the labels, thereby reducing the convergence speed and possibly affecting the performance [2, 18]. (2) Edge-cut based algorithm in general and METIS algorithm in particular is not guaranteed the optimality of the clustering results. Moreover, due to multilevel bisection properties, METIS needs to specify the number of clustering to extract from the graph, which also can affect the quality of clustering results [19].

1.3. Objectives and Contributions

The goal of this work is to improve the performance of the cluster-GCN model. More specifically, this work focuses on the sampling phase where the graph(s) are divided into many smaller subgraphs. The thesis has 2 contributions as follow:

- Testing the efficiency of Leiden algorithm in graph-wise sampling phase of Cluster-GCN architecture
- Suggestions in adding constraints (e.g., minimum/maximum community size, overlapping community) to Leiden algorithm to improve efficiency of Cluster-GCN

1.4. Organization

This thesis is structured as follows. Chapter 2 introduces and discusses challenges with the dataset. Chapter 3 details our strategy which includes implementation of Cluster-GCN architecture, Leiden algorithm and proposed constraints for Leiden algorithm. Lastly, chapter 4 presents experiment results, conclusion of the thesis and suggestions for further study in the future.

Chapter 2 Data exploration

2.1. Dataset introduction

In this study, we evaluate our proposed improvement for Cluster-GCN with **ogbn-proteins** dataset. This is a protein-protein association network whose proteins are collected from 8 species [20, 21]. The dataset contains an undirected, weighted graph. In particular, each edge describes the biologically relationship between a pair of proteins. In computational setting, edge features are represented by an 8-dimensional vector, where each dimension represents the degree of confidence that there will be a single type of relationship and takes on values between 0 and 1 (the closer the value to 1, the more certain we are that the relationship exists.). The graph statistics are given in Table 1.

Additionally, the OGB team also provides researchers with data loaders as packages in Python that automate downloading and pre-processing of the datasets. Accordingly, data are divided into train, test, and validation sets. Distribution of node species ID are given in Figure 4 below.

Table 1: Statistics of currently-available ogbn-proteins dataset.

Dataset	#Graphs	#Node	#Edge	#Labels	#Edge Features
ogbn-proteins	1	132534	39561252	112	8

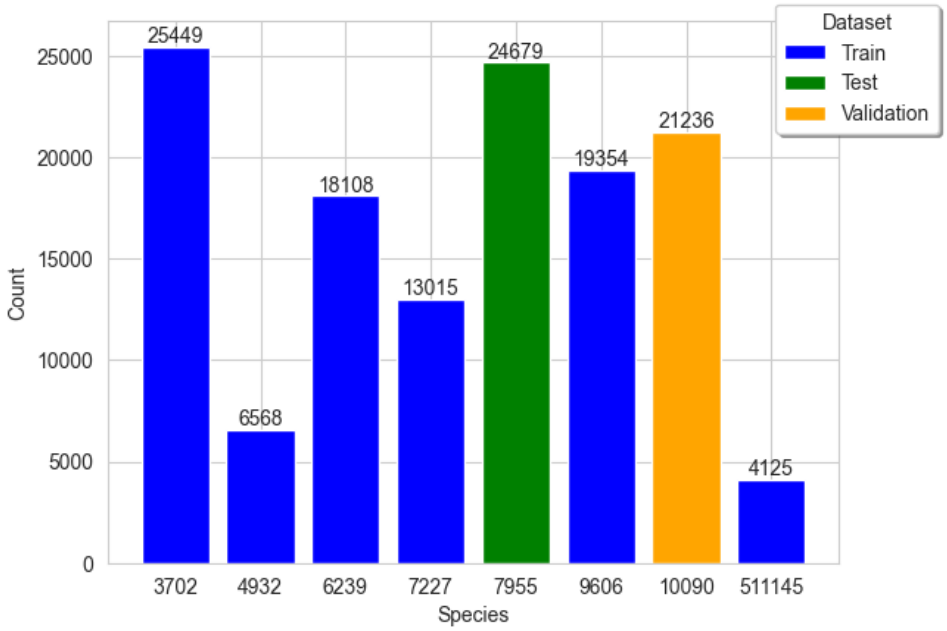


Figure 4: Distribution of node species among Train, Validation and Test set

2.1.1. Prediction task

The challenge is in predicting the likelihood that protein functions would be present in a multi-label binary classification scenario, where there are a total of 112 labels to predict (corresponding to 112

functions).

2.1.2. Dataset splitting

The dataset is splitted into 3 sets: training set, validation set and test set as provided in the loader package. According to the author, this enables researchers to make generalized evaluations of the model across different species.

2.1.3. Discussion

There are two points we want to discuss in this part of the thesis:

Firstly, as written in the paper [22], because the authors' choice of species ID one-hot encodings as node features, the ogbn-proteins dataset actually lacks features for input nodes. Instead, it has edge features, which are useful. Therefore, the dataset raises an intriguing research question of how to utilize edge information in a more complex manner rather than naive averaging. The challenge is to efficiently manage the huge amount of edge features on GPU at scale, which may need sophisticated graph splitting that utilizes edge weights.

Secondly, although we acknowledge that this dataset has a relatively small number of nodes and can be easily handled by GPUs, we still want to experiment with the Cluster-GCN to assess its effects in an environment with limited resources that the majority of undergraduate students can afford.

2.2. Edge feature learning: Aggregate edge features to nodes

As introduced in the paper [22], all edges contain valuable information. Those are important and should be considered during training to boost the prediction capacity of the model. However, Cluster-GCN [2] architecture does not have the function to handle edge features but node features; therefore, before the sampling process takes place, instead of using the speciesID, the model constructs a set of node features by aggregating edge features of each node's entire neighbors. This step can be formulated as follow:

$$\boldsymbol{x}_i = aggr_{j \in \boldsymbol{\mathcal{N}}(i)}(\boldsymbol{e}_{ij})$$

where $\boldsymbol{x}_i$ is the feature of node i, $aggr(.)$ is the differentiable and permutation invariant functions such as add, mean or max, $\boldsymbol{\mathcal{N}}(i)$ is the set of node's i neighbors and $\boldsymbol{e}_{ij}$ is the edge feature between node i and node j. By doing this, each node feature can contain some information of the whole complete graph. Figure 5 illustrates the edge aggregation process.

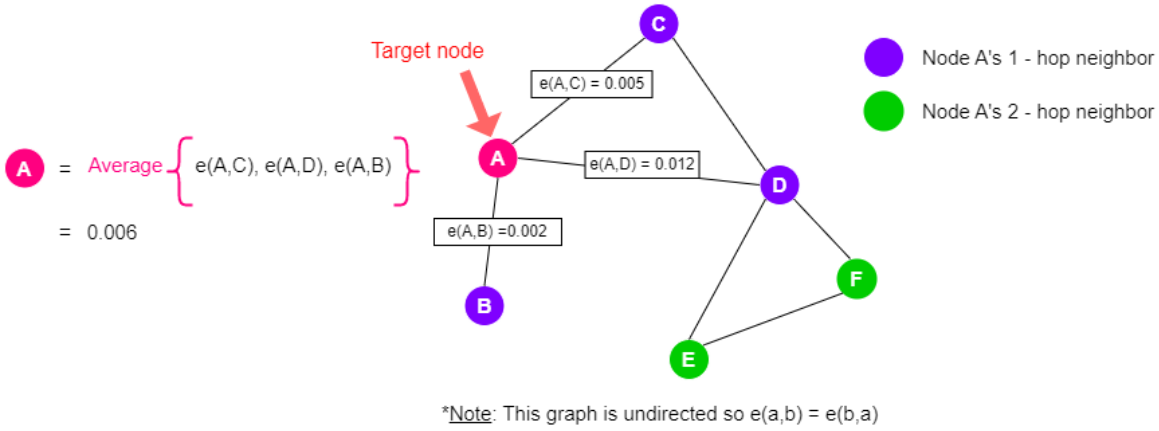


Figure 5: The illustration of edge feature learning with mean aggregation

Chapter 3 Methodology

3.1. Cluster-GCN

To be able to bring the most accurate comparison results, this work uses a baseline model based on the network architecture of the above Cluster-GCN [2]. The components of the Cluster-GCN model are described in detail in each section below.

3.1.1. Graph-wise Sampling

Both of the sampling methods mentioned in section 1.2.1 suffer from memory inefficiency or slow convergence speed. To tackle these problems, Chiang et al [2] proposed another sampling method called Graph-wise Sampling, which extract mini-batch in the graph level instead of the node level. Formally, the graph $\boldsymbol{\mathcal{G}} = \{\boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}}\}$ can be divided into c subgraph $\bar{\boldsymbol{\mathcal{G}}} = \left[\boldsymbol{\mathcal{G}}_1, \dots, \boldsymbol{\mathcal{G}}_c\right]$ by split $\mathcal{V}$ into c group of nodes: $\boldsymbol{\mathcal{V}} = \left[\boldsymbol{\mathcal{V}}_1, \dots, \boldsymbol{\mathcal{V}}_c\right]$. The adjacency can be re-formulate as:

$$\mathbf{A} = \bar{\mathbf{A}} + \boldsymbol{\Delta} = \begin{bmatrix} \mathbf{A}_{11} & \dots & \mathbf{A}_{1c} \\ \dots & \dots & \dots \\ \mathbf{A}_{c1} & \dots & \mathbf{A}_{cc} \end{bmatrix} \tag{3.1}$$

where $\bar{\mathbf{A}}$ is a diagonal matrix contain all edge inside a subgraph: $\bar{\mathbf{A}} = [\mathbf{A}_{11}, \dots, \mathbf{A}_{cc}]$ and $\boldsymbol{\Delta}$ is an off-diagonal matrix containing all edges between subgraphs. To support the idea of the influence of edges in the same batch on mini-batch, different graph clustering algorithms can be applied to retain more edges within a batch.

3.1.2. Vanilla Cluster-GCN architecture

Cluster-GCN [2] use Metis algorithms [8] to partition whole graph $\mathcal{G}$ into c sub-graph $\bar{\boldsymbol{\mathcal{G}}} = \left[\boldsymbol{\mathcal{G}}_1, \dots, \boldsymbol{\mathcal{G}}_c\right]$. The benefit of this step is that GCNs can be decomposed into batches for mini-batch training. The propagation flow from equation (1.5) can be rewrite as:

$$\mathbf{H}^{(l+1)} = f\left(\mathbf{H}^{(l)}, \mathbf{A}\right) = \sigma\left(\bar{\mathbf{A}}'\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right)$$

$$= \begin{bmatrix} \sigma\left(\bar{\mathbf{A}}'_{11}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \\ \dots \\ \sigma\left(\bar{\mathbf{A}}'_{cc}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \end{bmatrix} \tag{3.2}$$

where $\bar{\mathbf{A}}'_{ii}$, $0 \leq i \leq c$ is the normalized and regularized version of i-th cluster adjacency matrix. By only considering links inside the cluster and ignoring between-clusters links, Cluster-GCN can prevent the neighborhood expansion problem of the previous sampling method (Figure 6).

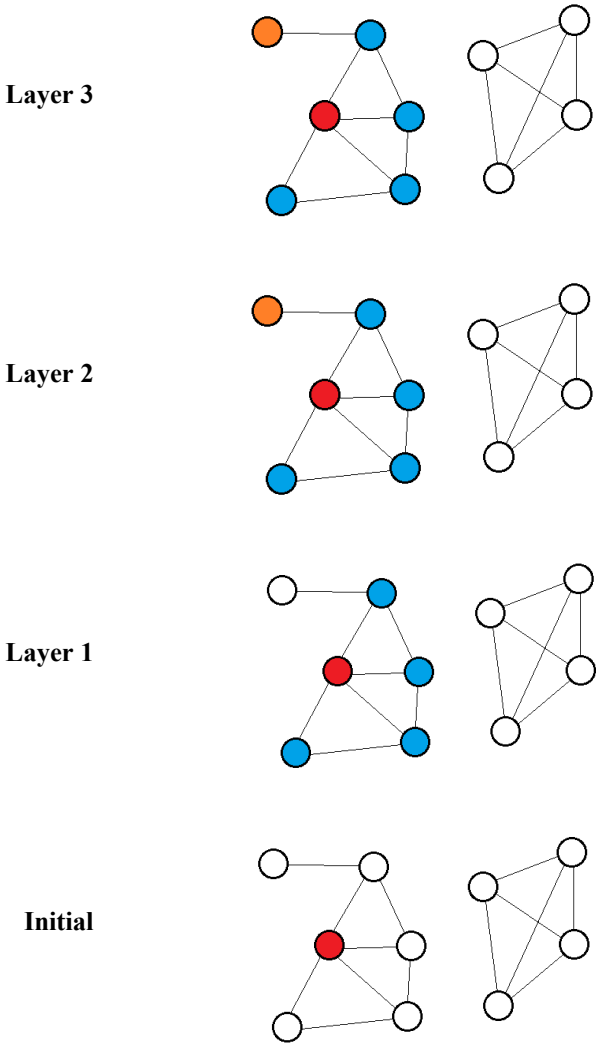


Figure 6: Propagation of Cluster-GCN. Red node is the starting node, blue node indicates 1-hop neighborhood, orange is 2-hop neighborhood and green is 3-hop neighborhood.

However, graph clustering algorithms only consider graph structure to split graphs into separate clusters, hence similar nodes will be in the same cluster. Therefore, the label distribution for each cluster can be different and skew toward each cluster, which affects the convergence speed and may result in worse performance than random clustering (Figure 7).

3.1.3. Stochastic Multiple Partitions

To tackle skewed label distribution between clusters, Chiang et al [2] proposed Stochastic Multiple Partitions. Specifically, graph $\mathcal{G}$ is clustered into very large number of cluster $N_c \gg c$ and then each batch is formed by randomly select a small number bs (batch size) clusters (Figure 8). Furthermore,

all edges between *bs* clusters that is selected: $\{\mathbf{A}_{ij} \mid i,j \in \boldsymbol{\mathcal{V}}_1, \ldots, \boldsymbol{\mathcal{V}}_{bs}\}$ is also added back to reduce the number of lost edges between clusters. This technique reduces variance across batches, therefore, it boosts the convergence speed and may increase performance.

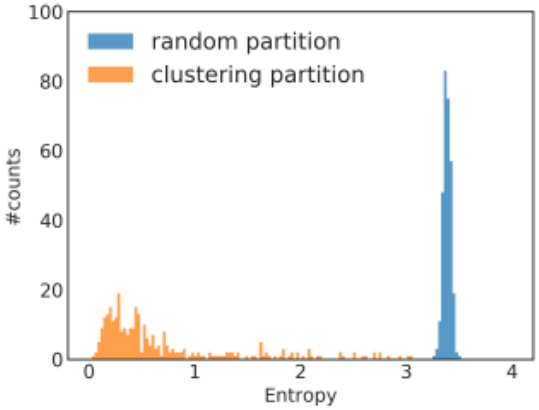


Figure 7: Histogram of entropy label distribution. Low label entropy indicates skew label distribution within each batch. As we can see, Metis clustering results in a low entropy compared to random clustering due to similar nodes that tend to be in the same cluster. Picture from paper [2]

3.1.4. Training deeper GCNs

For a small dataset, a small number of layers can cover the whole graph, which means that adding more layers is not helpful. However, for large networks, this may not be corrected. To deep dive into this, Cluster-GCN [2] investigated deeper GCNs for large networks by borrowing residual ideas from Resnet [23]. Formally, the propagation from (1.5) is modified by adding the hidden representations in *l*-th layer to the next layer:

$$\mathbf{H}^{(l+1)} = f\left(\mathbf{H}^{(l)}, \mathbf{A}\right) = \sigma\left(\mathbf{A}'\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) + \mathbf{H}^{(l)} \tag{3.3}$$

However, this strategy does not take the number of layers into account [2] since the coefficient of activations and the adding representation is equal. Intuitively, the neighbor nearby should contribute more than the nodes that are far away. Based on that, the representation from previous layers should have more weight than the following layers. Formally:

$$\mathbf{H}^{(l+1)} = f\left(\mathbf{H}^{(l)}, \mathbf{A}\right) = \sigma\left((\mathbf{A}' + \mathbf{I})\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) + \mathbf{H}^{(l)} \tag{3.4}$$

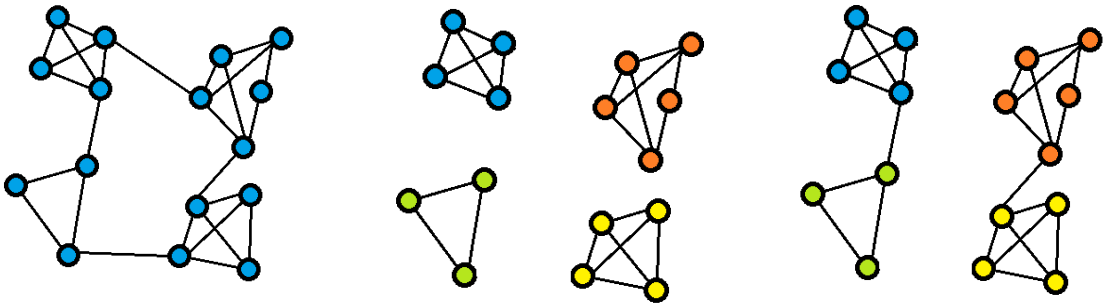
Furthermore, to normalized with respect to degree of each node, the term $\mathbf{A}'$ is modified to:

$$\hat{\mathbf{A}} = (\mathbf{D} + \mathbf{I})^{-1}(\mathbf{A} + \mathbf{I}) \tag{3.5}$$

With (3.5), the equation (3.4) is modified to:

$$\mathbf{H}^{(l+1)} = \sigma\left(\left(\hat{\mathbf{A}} + \lambda diag(\hat{\mathbf{A}})\right)\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) + \mathbf{H}^{(l)} \tag{3.6}$$

By changing propagation from (3.4) to (3.6), Cluster-GCN adopted the "diagonal enhancement" techniques [2] where the weight of the previous layer can be modified as a hyperparameter, which we can tune to build deeper GCNs and get better or even state-of-the-art performance for different problems.



a. Original graph b. Applied Metis [8] c. Forming batches

Figure 8: Illustration of stochastic multiple partitions with $N_c = 4$ and $bs = 2$. Note that in step c, each cluster is randomly selected to form a bigger cluster (blue cluster can be formed with orange cluster and green cluster can be formed with yellow cluster in different iterations)

3.2. Leiden community detection

Definition (Modularity [17]): Modularity is a metric used to assess how well nodes are assigned to communities. In other words, it measures how densely connected nodes within a community are compared to connection of those nodes in the whole graph. Modularity can be described as:

$$\begin{aligned} \boldsymbol{Q} &= \frac{1}{2m}\sum_{i,j}\left(\mathbf{A}_{i,j} - \gamma\frac{k_i k_j}{2m}\right)\delta(c_i, c_j) \\ &= \frac{1}{2m}\sum_{c=1}^{\mathbf{C}}\left(\boldsymbol{e}_c - \gamma\frac{K_c^2}{2m}\right) \end{aligned} \tag{3.7}$$

where $\boldsymbol{Q}$ is the modularity; m is the total number of edge in the graph; k_i is the sum of degrees of node i; δ is the Kronecker delta function (equal to 1 when $c_i = c_j$, 0 otherwise); $\mathbf{C}$ is the set of communities in the graph; $\boldsymbol{e}_c$ is the number of edge in the community c; K_c is the sum of degrees of the nodes in community c and $\gamma > 0$ is the resolution parameter. Intuitively, a community is normally a small group of nodes within a network, whose connection between nodes is extremely dense. This lead to a small value of K_c^2 and a large value of $\boldsymbol{e}_c$. Based on that, detecting community is become maximizing the different between $\boldsymbol{e}_c$ and K_c^2, or maximizing $\boldsymbol{Q}$. At this point, γ can be used to adjust the number of communities in the network. With higher value of γ, to maintain $\boldsymbol{Q}$, K_c^2 must be smaller, which lead to an increasing number of communities, while lower γ, in contrast, leads to

fewer communities.

Despite its clear definition, optimizing modularity is NP-Hard [24], therefore, many heuristic algorithms have been proposed [11, 12]. The Leiden community detection [12] is the one of the most popular modularity-base hierarchical clustering algorithms that is used for detecting communities (relatively dense groups) in a given graph. Specifically, Leiden takes advantages from smart local move algorithms to recursively merge communities by greedily optimizing modularity. Those processes can be divided into three phases: (1) Local moving of nodes; (2) Refinements of the partitions and (3) Aggregation of the network based on the refined partition, using the non-refined partition to create an initial partition for the aggregate network [12]. Each phase of the Leiden algorithm is discussed in the sections below. The full algorithm is described in pseudo-code in [25]. Figure 9 is an illustration of this algorithm.

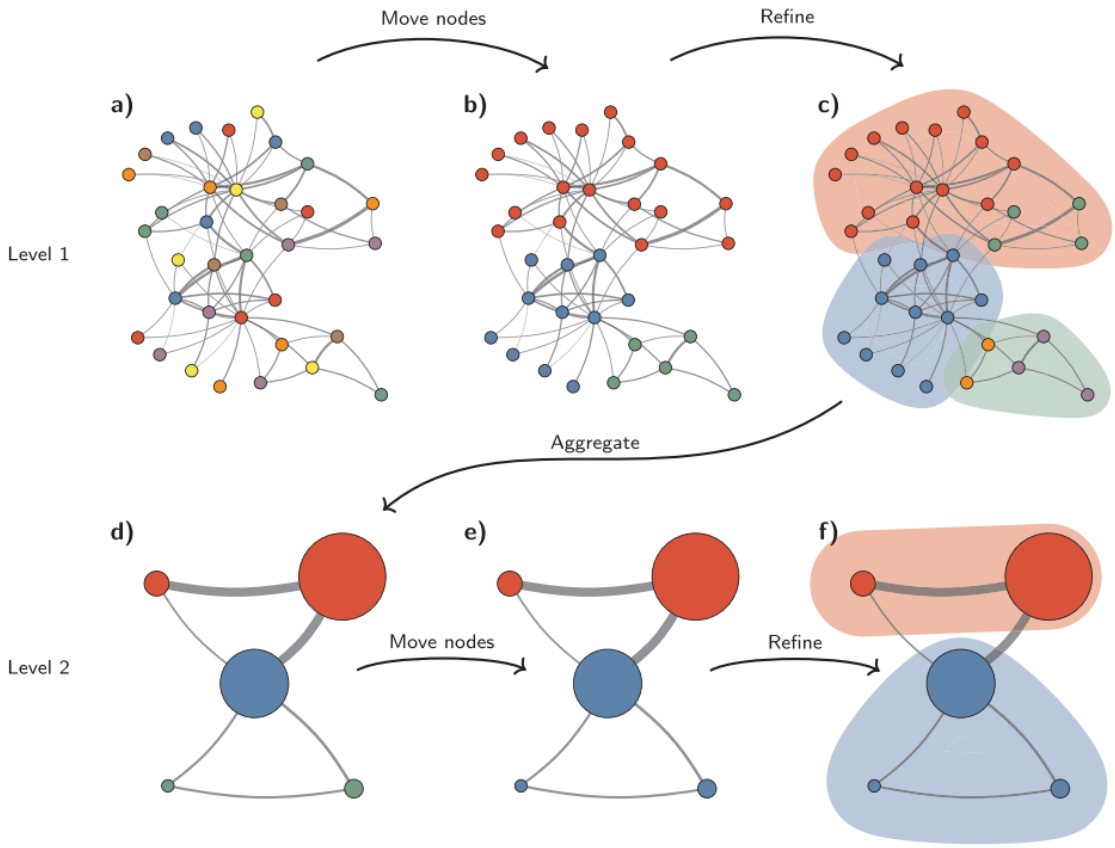


Figure 9: Illustration of Leiden algorithm. Those steps are repeated until modularity cannot improve further. Picture from paper [12].

3.2.1. Local moving nodes

Assume given graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ contain N nodes. The Leiden algorithm starts by assigning a different community for each node in the graph. Therefore, after the initialization, the number of communities is equal to the number of nodes in the graph, which is N. Then, for each node i, a smart local moving algorithm is applied. This algorithm consider each neighbor j of node i to compute the gain in modularity ($\Delta\mathcal{Q}$) if node i is removed from it community and placing it in the community of j:

$$\Delta \boldsymbol{Q} = \left[\frac{\Sigma_{in}+k_{i,in}}{2m} - \left(\frac{\Sigma_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \quad (3.8)$$

where Σ_{in} is the sum of the weights of the links inside the community of j; $k_{i,in}$ is the sum of the weights of the links from i to nodes in community of j and Σ_{tot} is the sum of the weights of the links within community of j. The first term (left side of minus sign) in equation (3.8) is the new modularity when node i is removed from it community and placing it in the community of j and the second term (right side of minus sign) is the old modularity before local moving is perform. However, by visiting all nodes until there is no more gain in modularity is wasteful due to nodes that are well-connected in their community cannot move to other communities. Therefore, Leiden speed-up local moving nodes by only visiting nodes whose neighborhood community has changed.

3.2.2. Refinement of partitions

Usually, after moving nodes, aggregation of the network is performed to merge all the nodes in the same cluster, like the Louvain algorithm [11]. However, it may create multiple subcommunities inside a community (Figure 10), which cannot be split further due to the property of agglomerative hierarchical clustering [12]. Therefore, to tackle subcommunity situation, Leiden maintain refinements partition $\boldsymbol{\mathcal{P}}_{refined}$ of original partition $\mathcal{P}$. Specifically, after Local moving nodes is performed, graph is divided into multiple partition, which is $\mathcal{P}$, however, there may be existed some community which contain multiple subcommunity inside it. The idea of $\boldsymbol{\mathcal{P}}_{refined}$ is to further divide those communities into multiple smaller communities (Figure 9c). First, each node is assigned to different communities in $\boldsymbol{\mathcal{P}}_{refined}$. Each communities $c_i \in \boldsymbol{\mathcal{P}}$ is then consider to be divided by extract all well-connected nodes v_{well}:

$$\boldsymbol{\mathcal{R}} = \left\{v_{well} \,\middle|\, v_{well} \in c_i, E\left(v_{well}, c_i - v_{well}\right) \geq \alpha \left\|v_{well}\right\| \times \left(\left\|c_i\right\| - \left\|v_{well}\right\|\right)\right\} \quad (3.9)$$

where $E(C, D) = |\{(u, v) \in E(\boldsymbol{\mathcal{G}}), u \in C, v \in D\}|$, α control how well-connected a node is, in the community. After that, each well-connected node is assigned randomly to communities $c^* \in \boldsymbol{\mathcal{P}}_{refined}$ based on a probabilities:

$$Pr(c^* = c') \sim e^{\frac{1}{\theta}\Delta \boldsymbol{Q}(v \to c')} \text{ if } \Delta \boldsymbol{Q} > 0 \text{ else } 0 \text{ for } c' \in \boldsymbol{\mathcal{P}}_{refined} \quad (3.10)$$

where $\frac{1}{\theta}$ act as normalization term and $\Delta \boldsymbol{Q}(v \to c')$ is the gain in modularity if node v is placing in community c'. However, cluster in $\boldsymbol{\mathcal{P}}_{refined}$ is initialized with a singleton partition, therefore, if all partitions c' are considered in equation (3.10), bad communities still happen. To get rid of this problem, Leiden defines the set of all well-communities inside communities $c \in \boldsymbol{\mathcal{P}}$:

$$\boldsymbol{\mathcal{T}} = \left\{c' \,\middle|\, c' \in \boldsymbol{\mathcal{P}}_{refined}, c' \subseteq c, E(c', c - c') \geq \alpha \left\|c'\right\| \times \left(\left\|c\right\| - \left\|c'\right\|\right)\right\} \quad (3.11)$$

Hence, $c' \in \boldsymbol{\mathcal{P}}_{refined}$ in equation (3.10) is changed to $c' \in \boldsymbol{\mathcal{T}}$, which means only dense subcommunities should be considered for further division and all communities are guaranteed to be subpartition α-dense.

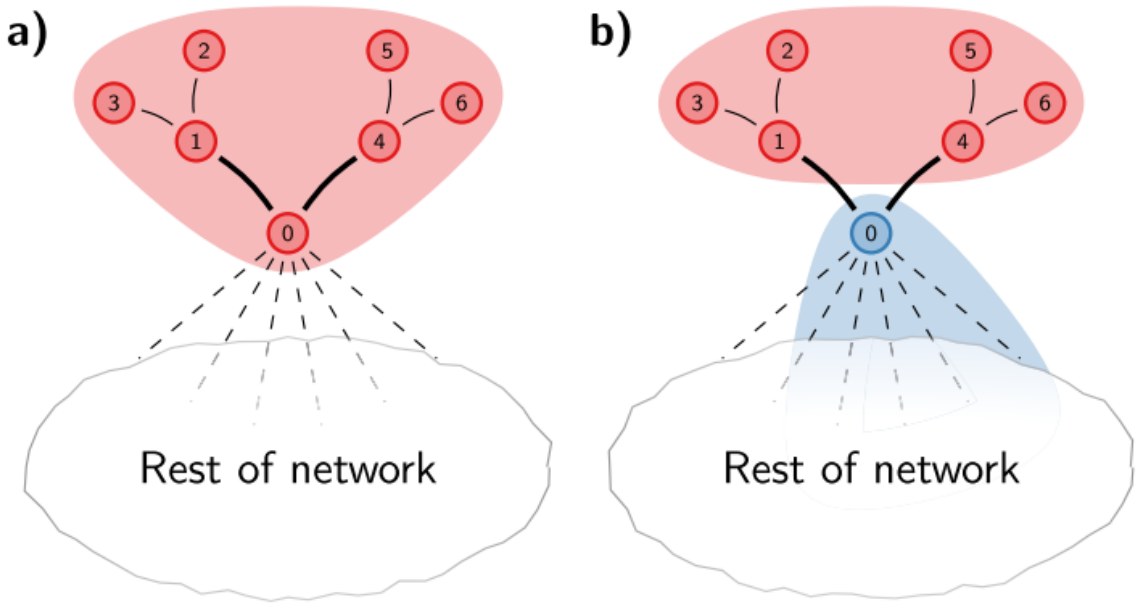


Figure 10: Illustration of a bad community created by the Louvain algorithm. After node 0 being moved to other communities, the red community contains 2 subcommunities, which are disconnected. Picture from [12]

3.2.3. Aggregation of the network

The final phase of the Leiden algorithm is building a new network $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}'\}$ whose nodes are communities found by the previous phase. Specifically, all node's weights within a community are aggregated together to form a weight for new nodes in the new network. The edge set of the new network is defined as:

$$\mathcal{E}' = \left\{(C, D) \,\middle|\, (u, v) \in E(\mathcal{G}),\ u \in C \in \mathcal{P}_{refined},\ v \in D \in \mathcal{P}_{refined}\right\} \tag{3.12}$$

which can be understood as the edge between communities. By creating the aggregate network based on $\mathcal{P}_{refined}$ rather than $\mathcal{P}$, the Leiden algorithm has more room for identifying high-quality partitions [12]. However, the initial partition for the aggregate network is based on $\mathcal{P}$. Specifically, after the new network $\mathcal{G}'$ is generated, the partition $\mathcal{P}$ is maintained corresponding to $\mathcal{G}'$ and act as a initialization for next iteration of the Leiden algorithm:

$$\mathcal{P} = \{\{v \,|\, v \subseteq C, v \in \mathcal{V}'\} \,|\, C \in \mathcal{P}\} \tag{3.13}$$

3.3. Constraint Leiden algorithm

By applying the maximizing modularity approach, Louvain and Leiden do not need to specify the number of communities to extract from the graph and all communities are guaranteed to be locally optimality assigned [12]. Furthermore, all communities extracted by Leiden algorithm are guaranteed to be subpartition α-connected. Due to this property, Leiden convincingly outperforms the Louvain algorithm [11], which has been tested and shows competitive results or even better compared to METIS algorithm [26, 27]. However, when Leiden algorithm is used as a sampling phase for cluster-GCN architecture, it still has the following limitations: (1) Size disparity among communities due to properties of agglomerative hierarchical clustering; (2) Intrinsic limitation of modularity, which is resolution limit. Therefore, Leiden is unable to detect small communities that are smaller

than a certain scale and (3) Skew the distribution of the labels due to properties of community detection, detailed in 1.2.2. In order to overcome these limitations, this work proposed adding constraints such as minimum/maximum community size and overlapping communities for Leiden algorithm to generate better subgraphs for mini-batch training. Each constraint is described in each section below.

3.3.1. Maximum and minimum community size

Maximum community size. For real-world dataset, the nodes are not evenly distributed among communities. Moreover, the Leiden algorithm takes advantage of the agglomerative hierarchical clustering to optimize modularity. Therefore, it would probably produce communities of vastly diverse sizes, among them, there may be communities that are much larger in size than others (Figure 11), which cannot be fitted into GPU for mini-batch training. These communities can further apply the Leiden algorithm to be divided into multiple smaller communities, which can be fitted into GPU for training. Formally, if there is a community C_i whose size is larger than ct_{max}:

$$C = (C - C_i) \cup \{c \mid c \in Leiden(C_i)\} \tag{3.14}$$

where C is the set of community in graph $\mathcal{G}$ and $Leiden(.)$ is the set of community extracted by the Leiden algorithm.This process is repeated until C do not contain any community whose size is larger than ct_{max}. However, the resolution parameter α must be increased to be able to separate a community into many subcommunities, which lead to the second limitation mentioned above. To deal with this problem, this work proposed a second constraint that controls minimum nodes in each community based on edges between them instead of using modularity. The details for this constraint will be discussed below.

Minimum community size. One of the advantages of the Leiden algorithm compared to Louvain is Leiden guaranteed that all communities are α-dense, implies that individual nodes are well connected to their community [12]. However, turning α is not an easy work and can lead to singleton partitions or partitions with a small number of nodes. Therefore, to get rid of the situation where Leiden creates small partitions, this work proposed a constraint to control the minimum number of nodes in each community extracted by the Leiden algorithm.

Intuitively, minimum constraint can be implemented by merging communities whose size is smaller than ct_{min} using the same algorithm like the first phase of the Leiden algorithm (section 3.2.1). However, for small communities, the probability that two separate communities can be merged is very low. Therefore, the probability that any two communities are merged together is approximately the same and the community chosen in equation (3.10) will follow the uniform distributions. With those limitations of modularity, the merging process is done by using the number of edges between communities. However, if only consider the number of edges across communities, communities with larger numbers of nodes normally have a higher number of edges, leading to merging processes that mainly focus on larger communities. Therefore, to be able to use the number of edges instead of modularity, this work proposed a normalization term for the number of edges. Specifically, each community C_i that satisfy $|C_i| < ct_{min}$ is considered to compute the edge ratio with others:

$$ER(C_i, C') = \frac{E(C_i, C')}{|C_i + C'|} + eps \tag{3.15}$$

where the term $|C_i + c'|$ as act the normalization that is used to balance the number of edges across communities with different communities size and eps is a small floating point used to prevent zero ratio. On this basis, community C_i is randomly merged with community C' with probability:

$$Pr(C_i = C') \sim \frac{ER(C_i, C')}{\sum_{C_j \in (C - C_i)} ER(C_i, C_j)} \quad (3.16)$$

With the idea from equation 3.10, the randomness from equation 3.16 allows to exploit the diversity in the community of networks.

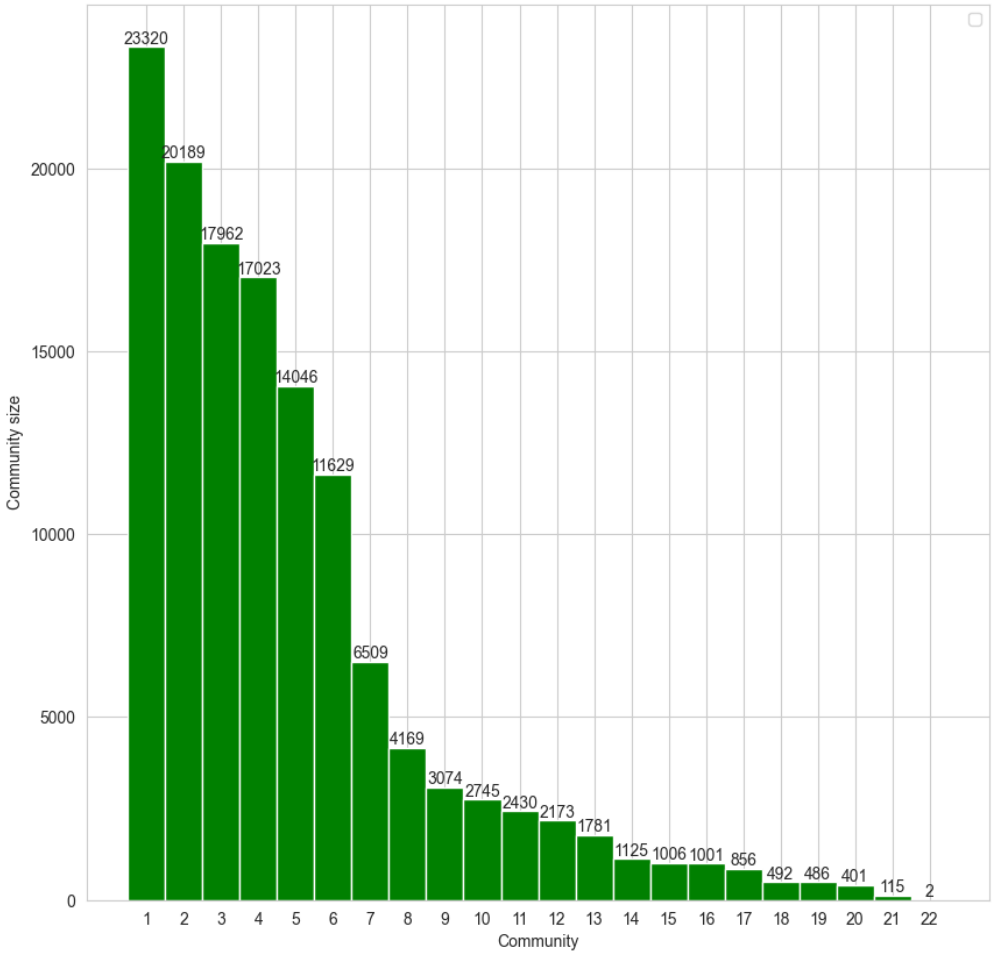


Figure 11: Size of communities extracted by the Leiden algorithm

Combine maximum with minimum constraint. To make mini-batch training effective, the amount of nodes between batches must be approximately the same. However, maximum constraint and minimum constraint only consider boundaries in one direction. Specifically, maximum constraint only guarantee that all communities have size smaller than ct_{max} while minimum constraint guarantee that all communities have size larger than ct_{min}. Intuitively, those two constraints can be used together so that all communities are guaranteed to have size within $[ct_{min}, ct_{max}]$. However, when both constraints are used but not related, it can lead to infinite loop problem due to the fact that merging communities process in minimum constraint can lead to a larger communities whose size is larger than ct_{max}. Those cluster, in maximum constraint, is further divided into smaller cluster whose size may be smaller than ct_{min} and etc. Therefore, to combine maximum with minimum constraint, equation 3.16 is modified so that merging communities is not allowed to create communities whose

size is larger than ct_{max}:

$$Pr(C' = C_i) \sim \frac{ER(C_i, C')}{\sum_{C_j \in (C - C_i)} ER(C_i, C_j)} \text{ if } |C_i + c'| \leq ct_{max} \text{ else } 0 \quad (3.17)$$

3.3.2. Overlapping community

Combining maximum and minimum community size constraints can handle limitations (1) and (2) discussed above. However, community detection algorithms in general and the Leiden algorithm [12] in particular only consider the structure of the graph (how nodes are connected together) to divide it into communities. Therefore, nodes in the same communities tend to be similar, leading to skew label distribution across batches. Despite the fact that Stochastic Multiple Partitions has reduced the effectiveness of this problem [2], it still suffers from skew node feature distribution across batches (green bar chart vs blue bar chart in Figure 12). To overcome skew node feature distribution, this work proposes overlapping communities to balance the node feature across batches, with the assumption that the subgraphs will be more similar to the original graph, thereby, it can boost the performance of the model.

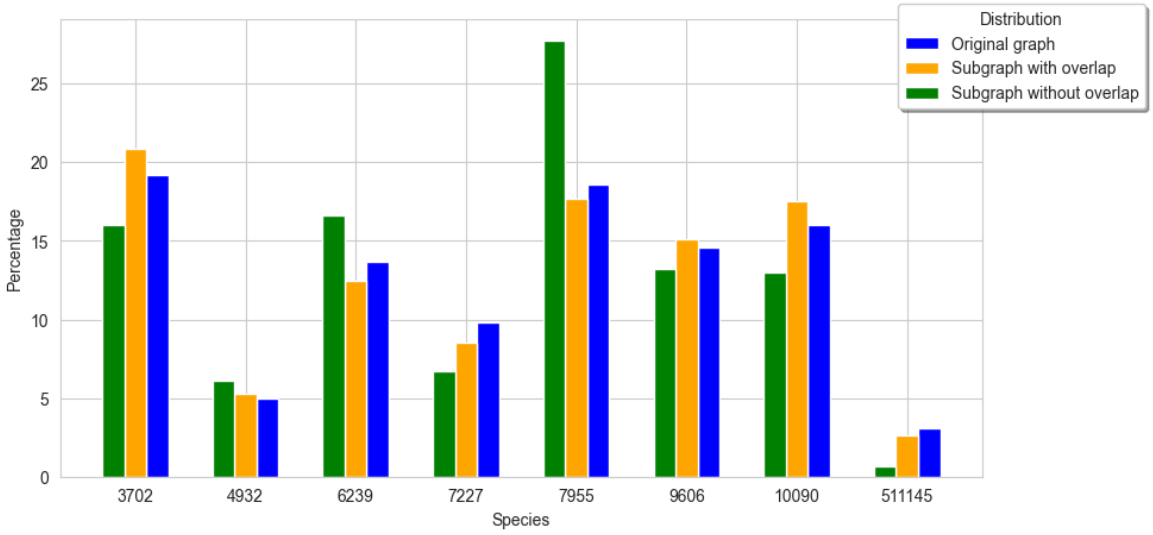


Figure 12: Difference between node features distribution of clusters generated by using the Leiden algorithm before and after using the overlapping community constraint. Green bar chart is the before and the orange chart is the after. Blue chart is the distribution of the whole graph.

As discussed above, a community extracted by community detection algorithms normally leads to a group of similar nodes. However, intuitively, the relationship between nodes not only indicates that their label is approximately similar, but also shows that, in some way, they have similar characteristics. Existing research [2, 18] only analyzes and handles skew between batches in the label distribution aspect but ignores the skew node feature across batches. Therefore, this thesis aims to balance the skewness across batches in the node feature aspect by modifying the Stochastic Multiple Partition phase from cluster-GCN [2]. Specifically, as discussed in section 3.1.3, original graph is divided into N_c community, which is very large. Instead of merging a number of communities together to form $bs \ll N_c$ subgraphs that is used for training mini-batch gradient descent, each community $c \in N_c$ is analyzed and divided into pre-defined $c_{feature}$ clusters based on its characteristics. In this work, dataset ogbn-proteins [22], whose nodes come from 8 species, which are

described as 8 dimension one-hot vectors, is used for experiments. Therefore, to keep it as simple as possible, instead of using clustering algorithms, $c_{feature}$ is set to 8, which is equal to the number of species that a node comes from. After that, assume f_i^c is the frequency of appearance of species *i* in the community *c*, each community is assigned with one from 8 species based on the most appearance of species within the community (which is i^*):

$$S(c) = \{i^*\} = \left\{argmax_{i \in S(\mathcal{G})}\left(f_i^c\right)\right\} \tag{3.18}$$

where $S(\mathcal{G})$ is the set of species in $\mathcal{G}$. Then, N_c community is turned into $N_{species} = \left\{N_1^s, \dots, N_8^s\right\}$, where $\sum_{N_i^s \in N_{species}} \left|N_i^s\right| = \left|N_c\right|$ and $N_i^s = \left\{c \mid argmax_{j \in S(\mathcal{G})}\left(f_j^c\right) = i,\ c \in N_c\right\}$ is the set of communities whose nodes mostly come from species *i*. From here, the randomness from section 3.1.3 is used. Specifically, assume the number of batches used for training mini-batch gradient descent is bs, for each $N_i^s \in N_{species}$, $\frac{\left|N_i^s\right|}{bs}$ communities are assigned to each batch, randomly. By this way, the balance of the label distribution is preserved, like Stochastic Multiple Partition in section 3.1.3. Furthermore, by narrowing the randomness within each species, each batch is equally distributed in the node feature aspect (orange bar chart vs blue bar chart in Figure 12). However, assigning a species for a community based on the species of mostly nodes within the community will ignore the case when there are two or more species whose frequency of appearance is approximately the same (Figure 13). Leading to the dividing of communities is skewed toward one specific species, despite the fact that the community may be evenly distributed among two or more species.

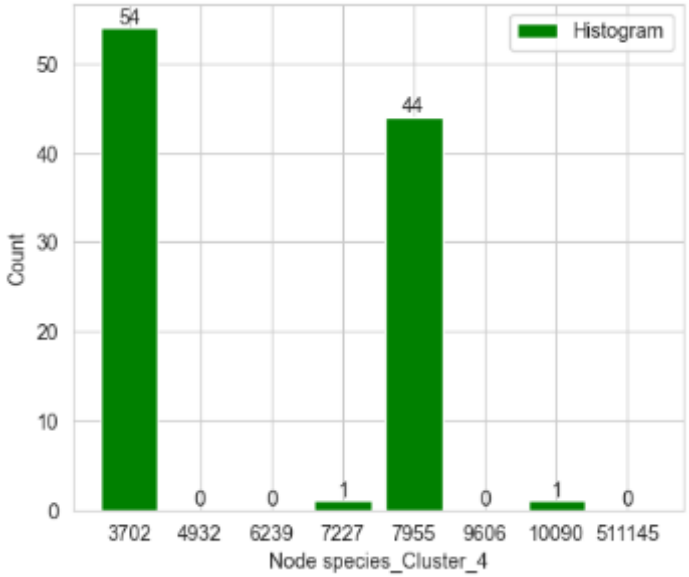


Figure 13: Communities with multiple species.

To solve the above problem, this work proposed the overlapping constraint: $0 \leq ct_{overlap} \leq 1$. Instead of assigning only one species for each community $c \in N_c$ like equation 3.18, community c can reach two or more species based on the largest frequency of appearance of species within community:

$$S(c) = \left\{ i \mid f_i^c \geq ct_{overlap} f_{i^*}^c \right\} \tag{3.19}$$

if $ct_{overlap}$ close to 1, each community tends to belong to only one species. In contrast, if $ct_{overlap}$ close to 0, each community tends to belong to all species, which reduces the effectiveness of dividing communities based on species. Based on the equation 3.19, $N_{species} = \left\{ N_1^s, \ldots, N_8^s \right\}$ is also changed, where $\sum_{N_i^s \in N_{species}} \left| N_i^s \right| \geq \left| N_c \right|$ and $N_i^s = \left\{ c \mid i \in S(c),\ c \in N_c \right\}$.

Chapter 4 Experiments and Conclusion

4.1. Experiments

To evaluate the effectiveness of constraints, experiments are performed in *ogbn-proteins* dataset [22], Moreover, the authors of ogbn-proteins dataset use standardized evaluator, which is average of ROC-AUC scores across the 112 tasks (detail in Appendix B) and a leaderboard to monitor state-of-the-art results [22]. Therefore, to ensure fairness, this work uses the assessment given by the authors of this dataset to evaluate the results. Finally, this work uses the random partition as a benchmark for comparison due to its superiority over the METIS algorithm [18].

4.1.1. Experiments setup

To understand the effectiveness of constraints, this work experiments on four different models:

r-Cluster-GCN. Cluster-GCN [2] with random partition. The architecture is described in Figure 14.

LeidenGCN. Cluster-GCN with the Leiden algorithm for community detection (blue block in graph-wise sampling phase in Figure 14).

b-LeidenGCN (ours). Bounded LeidenGCN, the LeidenGCN model with ct_{min} and ct_{max} constraints. The graph-wise sampling phase of b-LeidenGCN is described in Figure 15.

ob-LeidenGCN (ours). Overlapping and Bounded LeidenGCN architecture, the b-LeidenGCN with $ct_{overlap}$ constraint. The graph-wise sampling phase of ob-LeidenGCN is described in Figure 16.

Each model is fine-tuned to get the most objective comparison results. Common hyperparameters between architectures are listed in Table 2. Inspired by [18], to avoid the situation that lost edges are impossible to retrieve again during training, the community is generated at each epoch instead of just one before training. All models are implemented based on Pytorch Geometric [28]. Experiments are run on a single NVIDIA GeForce RTX 3070 8GB.

Table 2: Common hyperparameters between architectures

Hyperparameter	**Value**
Number of layers	{3,4,5,6,7}
Number of subgraphs	20
Edge feature learning type	Mean
Epochs	1000
Learning rate	0.005
Propagation dropout rate	0.5
Hidden channels	256

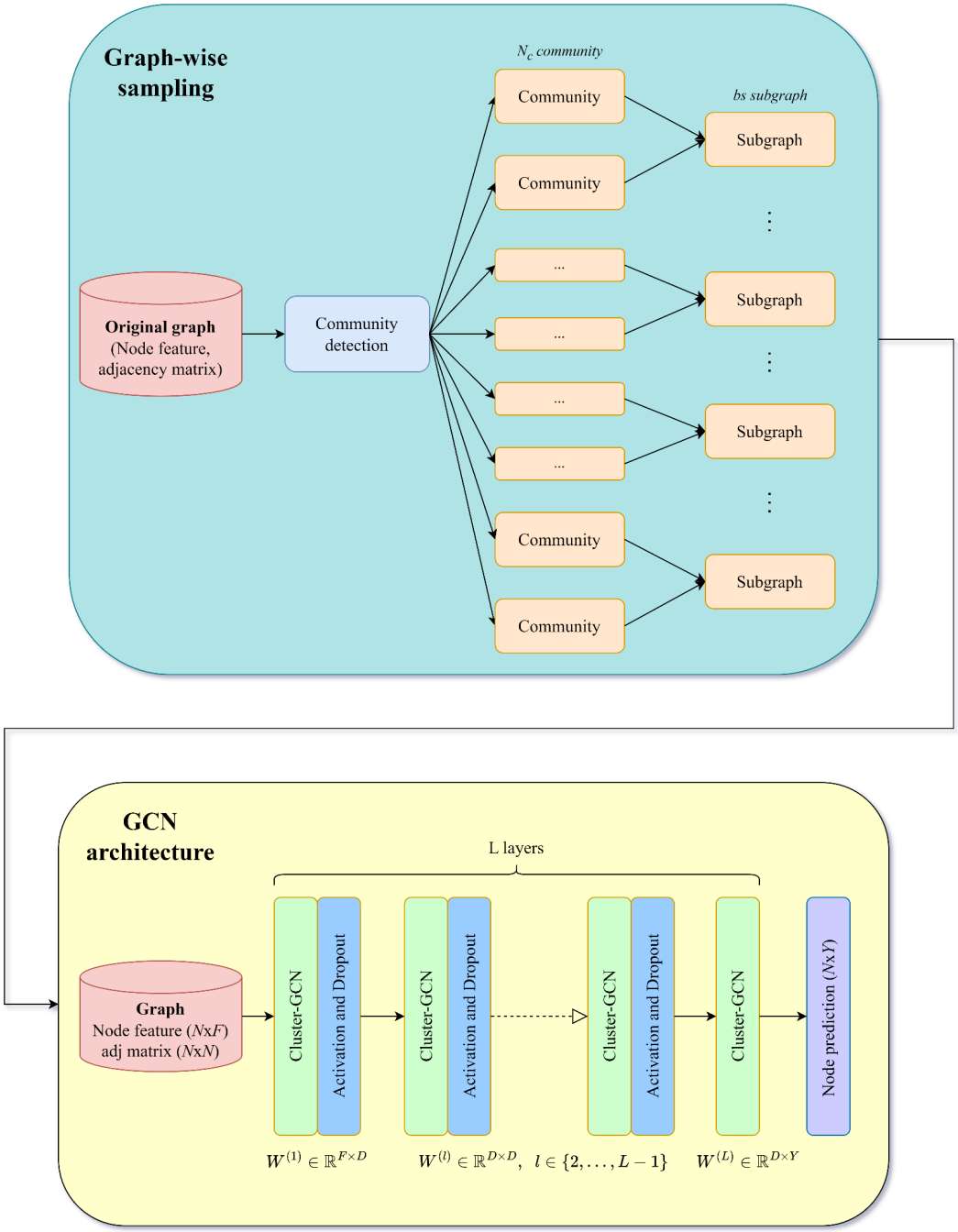


Figure 14: Baseline architecture. In Graph-wise sampling phase, communities are chosen randomly to merge together to form subgraphs. Community detection used in experiments is the Leiden algorithm. In GCN architecture, Cluster-GCN block is the message passing described in equation (3.6) and activation is the nonlinear function.

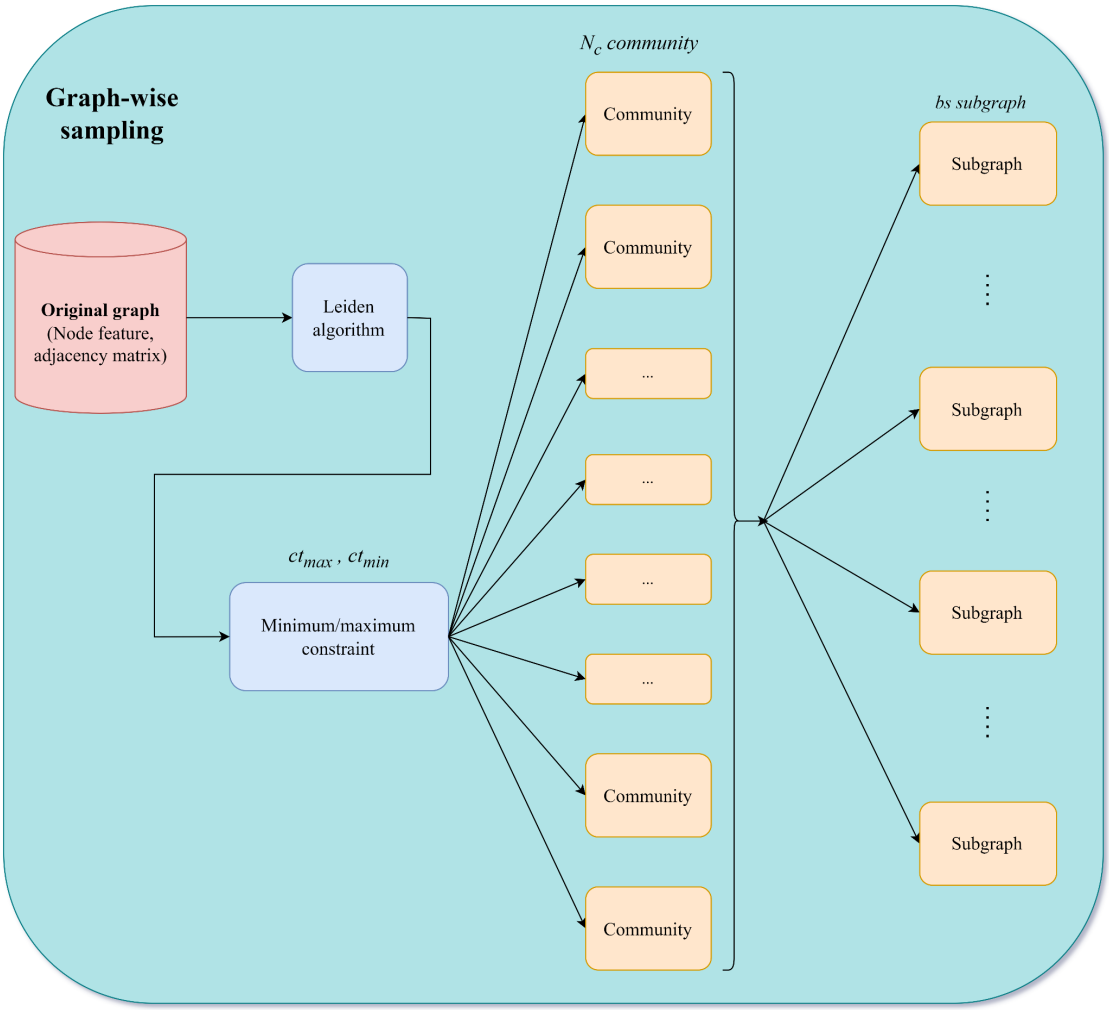


Figure 15: Graph-wise sampling using the Leiden algorithm with minimum/maximum community size constraints.

4.1.2. Results

Limitation of the Leiden algorithm. The Leiden algorithm has overcome the limitations of the METIS algorithm by automatically finding the optimal number of communities and guaranteeing the locally optimality of the partition. Therefore, when the number of communities is fixed according to the optimal number of communities extracted from the Leiden algorithm, this algorithm outperforms other algorithms in terms of the number of edges retained (Table 3), which is one of the reasons that lead to better results stated in [2]. Nevertheless, due to the properties of hierarchical agglomerative clustering, the number of nodes across communities is fluctuating significantly (Figure 11). Consequently, there may be some batches that are too large and cannot fit to gpu for training. If the Leiden algorithm is not refined properly, it cannot be used for the graph-wise sampling phase in the Cluster-GCN architecture.

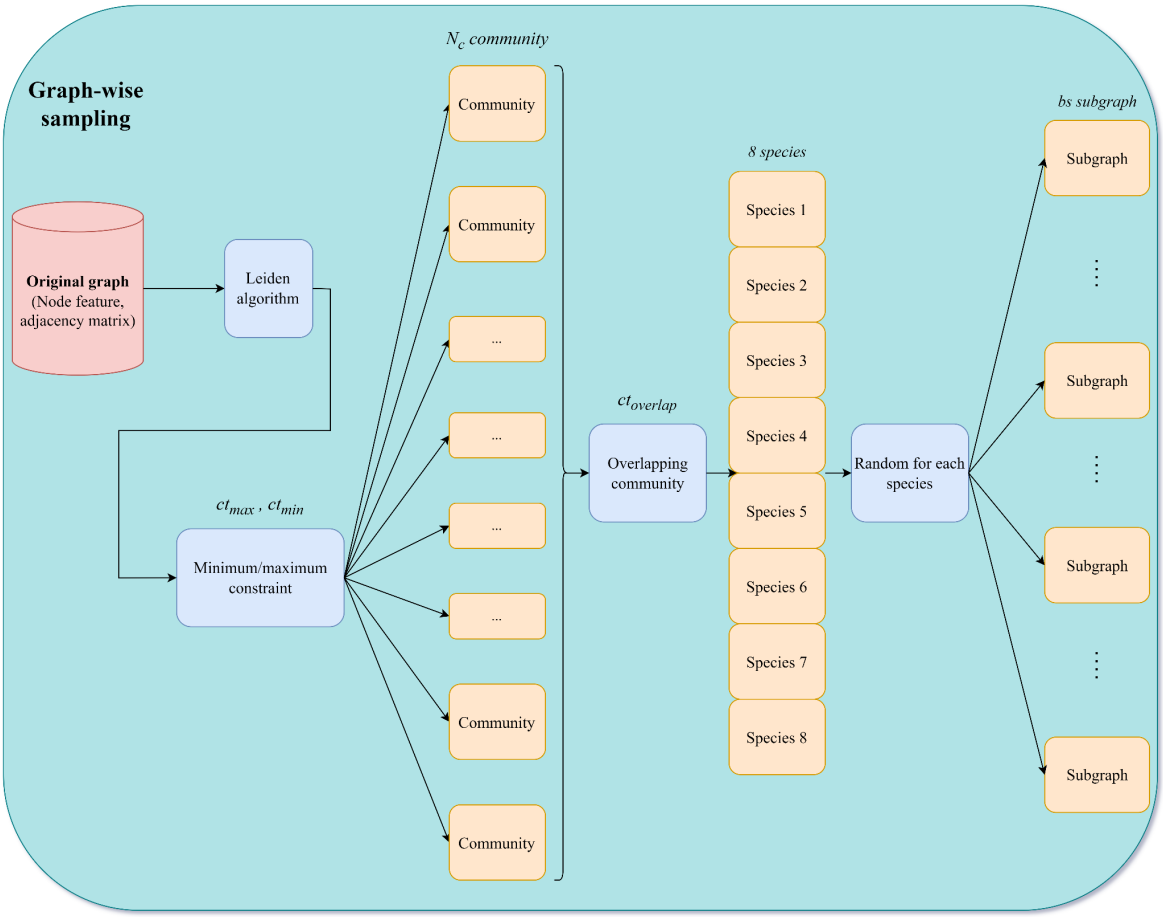


Figure 16: Graph-wise sampling using the Leiden algorithm with minimum/maximum community size and overlapping community constraints.

Table 3: Total number of edges retained using different algorithms. Results are the mean of 5 independent runs plus or minus the standard deviation.

Algorithm	**Total number of edges**
Random	1,798,910 ± 1,149.709
METIS [8]	25,407,758 ± 0
Leiden [12]	**33,055,514.5 ± 26,454.954**

Effect of maximum/minimum constraint. With ct_{min} and ct_{max} constraints, the number of nodes within each community is guaranteed to be in range $[ct_{min}, ct_{max}]$. Therefore, each batch is considered approximately equal in terms of size (Figure 17) and b-LeidenGCN is able to train with GPU. In terms of modularity, the Leiden algorithm outperforms METIS algorithm. However, continuing to split and force communities into the range $[ct_{min}, ct_{max}]$ lead to much smaller modularity compared to the original Leiden algorithm. From Table 4, the modularity of the Leiden algorithm reduce over 87% when $ct_{min} = 80$ and $ct_{max} = 100$. Nevertheless, this is a necessary trade-off to make communities size approximately even across batches. The random partition is the worst among 3 algorithms, whose modularity is almost zero due to the fact that each community is

generated randomly. In terms of ROC-AUC, from Table 5, it can be seen that b-LeidenGCN is 1.17% ahead of Cluster-GCN with the METIS algorithms. Although random partition is the worst in terms of modularity, r-Cluster-GCN with random partition outperform b-LeidenGCN and Cluster-GCN.

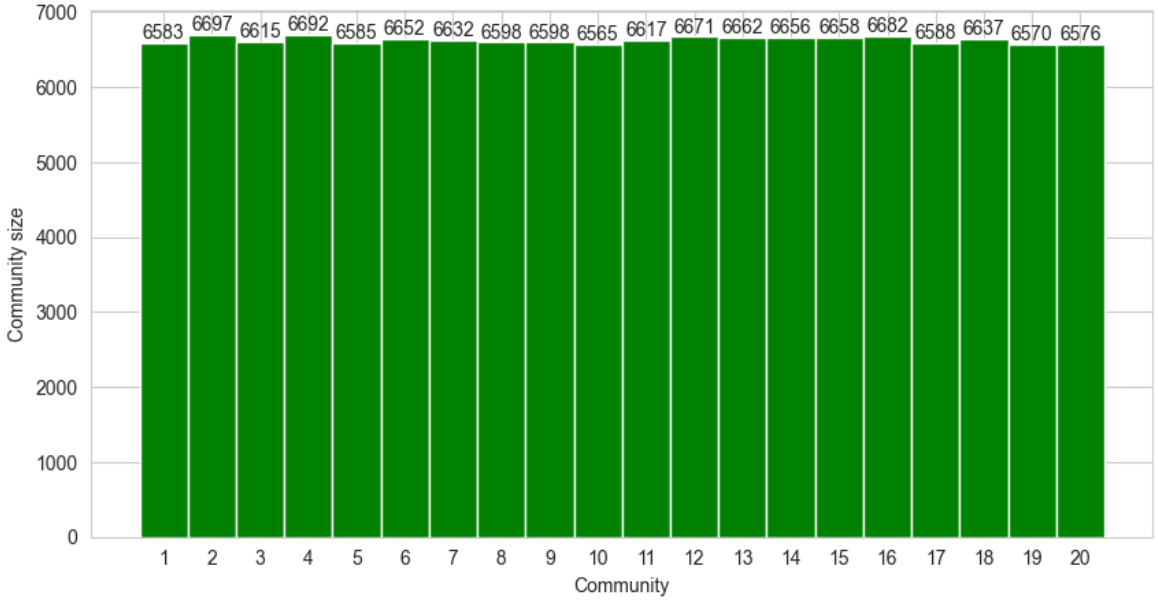


Figure 17: Number of nodes across batches after using maximum/minimum community size constraint. The number of batches is set to 20

Table 4: Modularity comparison with different constraints value and different algorithms.

Algorithm	**Modularity**
Random	-0.00003675
METIS [8]	0.08049441
Leiden [12]	**0.73213373**
Leiden [12], $ct_{min} = 80$ and $ct_{max} = 100$	0.09024661
Leiden [12], $ct_{min} = 200$ and $ct_{max} = 300$	0.18143729
Leiden [12], $ct_{min} = 400$ and $ct_{max} = 500$	0.19769391

Effect of overlapping constraint. By introducing $ct_{overlap}$ constraints, the node feature distribution across batches are adjusted to be similar (Figure 12). From Table 5, the performance of ob-LeidenGCN is improved 1.16% compared to b-LeidenGCN, which shows the effect of balancing the node feature distribution across batches. Moreover, although still behind r-Cluster-GCN 0.25%, the standard deviation across runs of ob-LeidenGCN is smaller, indicating that ob-LeidenGCN is the most stable model among models mentioned above. To see more clearly the effect of $ct_{overlap}$ constraint, ob-LeidenGCN is tested 2 times with different $ct_{overlap}$ values, detailed results are listed

in Table 6.

Table 5: Performance comparison between models. Result of Cluster-GCN is taken from paper [18] with mean and standard deviation of 10 runs, other models run 5 times. The constraints of b-LeidenGCN are set to $ct_{min} = 80$ and $ct_{max} = 100$. The number of layers is 3.

Algorithm	**ROC-AUC**
Cluster-GCN [2]	0.7513 ± 0.0044
r-Cluster-GCN	**0,7771 ± 0,0025**
b-LeidenGCN (ours)	0.7630 ± 0.0030
ob-LeidenGCN (ours)	0.7746 ± 0.0007

Table 6: The effect of $ct_{overlap}$ constraints. Constraint minimum/maximum community size are fixed to: $ct_{min} = 80$ and $ct_{max} = 100$. The number of layers is 3.

Algorithm	**ROC-AUC**
ob-LeidenGCN, $ct_{overlap} = 0.1$	0.7839313367
ob-LeidenGCN, $ct_{overlap} = 0.2$	0.7867266871
ob-LeidenGCN, $ct_{overlap} = 0.3$	0.7872546432
ob-LeidenGCN, $ct_{overlap} = 0.4$	0.788294836
ob-LeidenGCN, $ct_{overlap} = 0.5$	**0.7943720784**
ob-LeidenGCN, $ct_{overlap} = 0.6$	0.7875076298
ob-LeidenGCN, $ct_{overlap} = 0.7$	0.7894462016
ob-LeidenGCN, $ct_{overlap} = 0.8$	0.7910601422
ob-LeidenGCN, $ct_{overlap} = 0.9$	0.7913429585

Deeper models. As discussed in Chapter 3, node features are generated by aggregation of neighbor's edge features. Consequently, node features without any GCNs layer still contain a lot of information and this may be one of the reasons that r-Cluster-GCN, with the least number of edges, is the best method in Table 5. Moreover, community detection algorithms' ability to retain a large number of edges is also not exposed if the number of layers is set to be small. To clarify this, all models are trained with a larger number of layers, the results are listed in Table 7. When the number of layers is increased from 3 to 5, ob-LeidenGCN performance gets better and better, with a growth rate

approximately 1% per layer. Except for the case when the number of layers is 3, ob-LeidenGCN has the best performance among mentioned models. The peak is achieved when the number of layers equals 5, ob-LeidenGCN archives 79.4%, which is almost 1% better than r-Cluster-GCN. For both models, when the number of layers is larger than 5, it fails to converge and results in a loss of roc-auc. As stated in [2], the optimization for deeper GCNs becomes more difficult and the message passing needs to be refined so that the information from the first few layers is not impeded.

Table 7: Comparison of stacking deeper models. The constraints are fixed to $ct_{min} = 80$, $ct_{max} = 100$ and $ct_{overlap} = 0.5$. The results are the mean of 5 independent runs. The best results for each number of layers is bolded.

	ROC-AUC				
Algorithm	3 layers	4 layers	5 layers	6 layers	7 layers
r-Cluster-GCN	**0.7771**	0.7833	0.7842	0.7643	0.7597
ob-LeidenGCN (ours)	0.7746	**0.7848**	**0.7940**	**0.7864**	**0.7780**

4.2. Conclusion

This thesis proposes using Leiden algorithm with additional constraints to leverage the performance of Cluster-GCN. The results show that our approach achieves better classification outcome than random partition, which previously has been shown to outperform METIS. This demonstrated that fine tuning the graph-wise sampling step can improve model performance to some extent. We believe that this work suggests a direction for building more complex and deeper architectures.

In the future, we can expand the scope to a variety of community detection algorithms to analyze the effect of each algorithm on model performance. Furthermore, we aim to experiment with more datasets from different domains to demonstrate the effect of adding constraints. In order to do that, a generalization for the overlapping constraint needs to be studied.

Appendix

A. Proteins and representing protein data with graphs

Proteins data

Proteins play crucial roles in almost every important biological process by physically interacting with other proteins [29]. Therefore, to make significant contributions to biomedicine and pharmaceuticals, researchers must understand life at the molecular level, starting with correct annotations of protein functions. However, traditional methods to classify those functions are expensive and difficult to scale up to accommodate the vast amount of sequence data [30]. Fortunately, the advent of low-cost, high-throughput sequencing techniques and information technology makes the verification of a function become more accessible, especially when scientists need to work with newly discovered proteins [31].

Nonetheless, there are several reasons why computational biology faces numerous difficulties [32]. First, protein data, or biomedical data in general is often high-dimensional but sparse by nature. This is in contrast with large datasets in other domains such as social networks in computational social science and computer vision, which typically contain high-quality data. Second, biomedical data is frequently inaccurate and skewed due to technological, environmental, and physical limitations and especially biases in study design. Third, there are biological dynamics such as the evolution of cancer cells, bacteria, and viruses against drugs [33] that can lead to poor performance in computing models.

Representing protein data with graphs

Graphs naturally appear in the bioinformatics domain because of its capacity in capturing the structural relations of data. In a protein-protein association network, nodes represent proteins and edges signify various kinds of physiologically significant relationships such as physical interactions, co-expression, or homology [20]. Thus, traditional machine learning techniques that are based on statistical analysis cannot be directly applied for the computational tasks on graphs. More specifically, this domain often involves large-scale graphs with billions of edges or a dataset with millions of graphs. Recently, deep learning has proven its capabilities in representation learning, which has greatly advanced research in biology. Therefore, bridging deep learning with graphs will result in more powerful knowledge discovery, such as drug and material discovery. With that idea, the protein functions classification task becomes a single node property prediction task in the graph mining subfield.

B. Evaluation metric

To measure the performance of our model, we can count on an **ROC-AUC** Curve. The **ROC** (Receiver Operating Characteristics) - **AUC** (Area Under the Curve) is one of the most important evaluation metrics for checking any model's classification capability, which can be helpful in visualizing the performance of a binary classification problem.

To be more specific, ROC is a probability curve and AUC represents the degree or measure of separability. By looking at the curve, researchers know the extent that the model can distinguish between classes. AUC can be considered as the likelihood that the model ranks a random positive example higher than a random negative example. The higher the AUC score, the better the model is at predicting 0 classes as 0 and 1 classes as 1. In our experiment, by analogy, higher AUC means more precise protein function prediction.

Defining terms used in AUC and ROC Curve.

- TPR (True Positive Rate) / Recall / Sensitivity

$$TPR = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

- Specificity

$$Specificity = \frac{True\ Negative}{True\ Negative + False\ Positive}$$

- FPR

$$FPR = 1 - Specificity\ or\ \frac{False\ Positive}{True\ Negative + False\ Positive}$$

Visualization.

The ROC curve is plotted with TPR against FPR where TPR is on the y-axis and FPR is on the x-axis. Besides, AUC measures the entire two-dimensional area underneath the entire ROC curve. Figure 18 gives an example of AUC-ROC visualization.

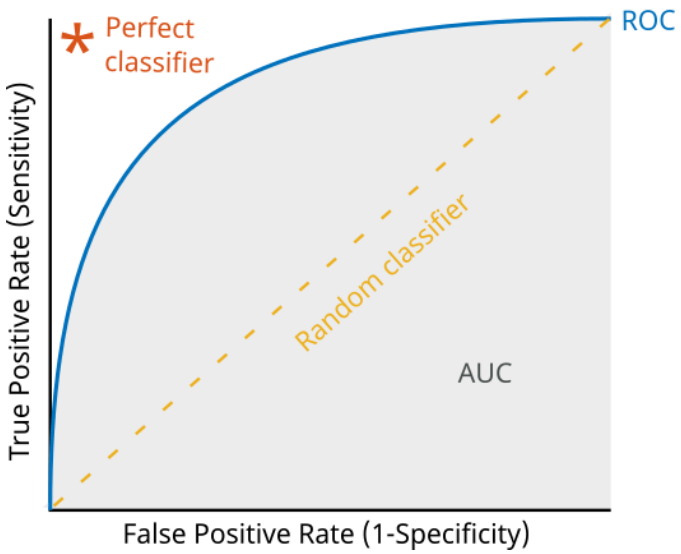


Figure 18: Example of a ROC-AUC curve image - by Mathworks.com

References

1. KIPF, Thomas N.; WELLING, Max. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 2016.
2. CHIANG, Wei-Lin, et al. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. 2019. p. 257-266.
3. HAMILTON, Will; YING, Zhitao; LESKOVEC, Jure. Inductive representation learning on large graphs. Advances in neural information processing systems, 2017, 30.
4. LI, Guohao, et al. Deepergcn: All you need to train deeper gcns. arXiv preprint arXiv:2006.07739, 2020.
5. LIU, Weile, et al. DCBGCN: An Algorithm with High Memory and Computational Efficiency for Training Deep Graph Convolutional Network. In: 2020 3rd International Conference on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE). IEEE, 2020. p. 16-21.
6. CHEN, Jie; MA, Tengfei; XIAO, Cao. Fastgcn: fast learning with graph convolutional networks via importance sampling. arXiv preprint arXiv:1801.10247, 2018.
7. CHEN, Jianfei; ZHU, Jun; SONG, Le. Stochastic training of graph convolutional networks with variance reduction. arXiv preprint arXiv:1710.10568, 2017.
8. KARYPIS, George; KUMAR, Vipin. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on scientific Computing, 1998, 20.1: 359-392.
9. LI, Guohao, et al. Training graph neural networks with 1000 layers. In: International conference on machine learning. PMLR, 2021. p. 6437-6449.
10. LUO, Man, et al. A Novel High-Order Cluster-GCN-Based Approach for Service Recommendation. In: Web Services–ICWS 2021: 28th International Conference, Held as Part of the Services Conference Federation, SCF 2021, Virtual Event, December 10–14, 2021, Proceedings. Cham: Springer International Publishing, 2022. p. 32-45.
11. BLONDEL, Vincent D., et al. Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment, 2008, 2008.10: P10008.
12. TRAAG, Vincent A.; WALTMAN, Ludo; VAN ECK, Nees Jan. From Louvain to Leiden: guaranteeing well-connected communities. Scientific reports, 2019, 9.1: 5233.
13. BRUNA, Joan, et al. Spectral networks and locally connected networks on graphs. arXiv preprint arXiv:1312.6203, 2013.
14. DEFFERRARD, Michaël; BRESSON, Xavier; VANDERGHEYNST, Pierre. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 2016, 29.
15. https://ai.plainenglish.io/graph-convolutional-networks-gcn-baf337d5cb6b
16. ZHANG, Si, et al. Graph convolutional networks: a comprehensive review. Computational Social Networks, 2019, 6.1: 1-23.
17. NEWMAN, Mark EJ; GIRVAN, Michelle. Finding and evaluating community structure in networks. Physical review E, 2004, 69.2: 026113.
18. XIONG, Chenxin. Deep GCNs with Random Partition and Generalized Aggregator. 2020. PhD Thesis.
19. SHIOKAWA, Hiroaki; ONIZUKA, Makoto. Scalable Graph Clustering and Its Applications. 2018.
20. SZKLARCZYK, Damian, et al. STRING v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. Nucleic acids research, 2019, 47.D1: D607-D613.
21. GENE ONTOLOGY CONSORTIUM. The gene ontology resource: 20 years and still GOing strong. Nucleic acids research, 2019, 47.D1: D330-D338.
22. HU, Weihua, et al. Open graph benchmark: Datasets for machine learning on graphs. Advances in neural information processing systems, 2020, 33: 22118-22133.
23. HE, Kaiming, et al. Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. p. 770-778.

24. BRANDES, Ulrik, et al. On modularity clustering. IEEE transactions on knowledge and data engineering, 2007, 20.2: 172-188.
25. https://static-content.springer.com/esm/art%3A10.1038%2Fs41598-019-41695-z/MediaObjects/41598_2019_41695_MOESM1_ESM.pdf
26. LIU, Yike; SHAH, Neil; KOUTRA, Danai. An empirical comparison of the summarization power of graph clustering methods. arXiv preprint arXiv:1511.06820, 2015.
27. XU, Hongteng; LUO, Dixin; CARIN, Lawrence. Scalable gromov-wasserstein learning for graph partitioning and matching. Advances in neural information processing systems, 2019, 32.
28. FEY, Matthias; LENSSEN, Jan Eric. Fast graph representation learning with PyTorch Geometric. arXiv preprint arXiv:1903.02428, 2019.
29. WEBER, Gregorio, et al. Protein interactions. New York: Chapman and Hall, 1992.
30. OUZOUNIS, Christos A., et al. Classification schemes for protein structure and function. Nature Reviews Genetics, 2003, 4.7: 508-519.
31. RADIVOJAC, Predrag, et al. A large-scale evaluation of computational protein function prediction. Nature methods, 2013, 10.3: 221-227.
32. ZITNIK, Marinka, et al. Machine learning for integrating data in biology and medicine: Principles, practice, and opportunities. Information Fusion, 2019, 50: 71-91.
33. BICKER, Joana, et al. Elucidation of the impact of p-glycoprotein and breast cancer resistance protein on the brain distribution of catechol-o-methyltransferase inhibitors. Drug Metabolism and Disposition, 2017, 45.12: 1282-1291.