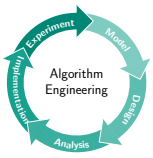

Bachelor Thesis

Improving Coarsening for Multilevel Graph Partitioning via Machine Learning

Simeon Hendrik Schrape

Date: 17. März 2025

Supervisors: Prof. Dr. Peter Sanders
M.Sc. Nikolai Maas
M.Sc. Daniel Seemaier
M.Sc. Kenneth Langedal

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Multilevel graph partitioning is a widely used heuristic for solving the $\mathcal{NP}$-complete BALANCED GRAPH PARTITIONING problem. It involves three phases: a coarsening phase that reduces the graph's size by contracting edges, an initial partitioning phase, and an uncoarsening phase that reverses the coarsening steps while refining the partition. The quality of the final partition depends on the effectiveness of the coarsening phase because bad contraction choices may destroy high-quality solutions on the coarser graphs. Current coarsening methods, like vertex clustering, typically rely solely on local connectivity, limiting their ability to capture the global structural properties of the graph. This thesis proposes a novel machine learning-guided coarsening algorithm that leverages a deep neural network to predict the likelihood of edges being cut in high-quality final partitions. These predictions guide the clustering process, thereby incorporating the global graph structure. Additionally, we investigate dimensionality and feature reduction experiments for reducing feature computation overhead without substantial quality loss.

Experiments demonstrate that the guided coarsening significantly improves partition quality, particularly for graphs with balanced edge-cut probability distributions. For such graphs, our approach achieves partitions with 2.87% fewer final edge cuts than the default configuration, which is almost the ground truth partition quality gain. However, partition quality does not reach its full potential on graphs with highly skewed probability distributions, highlighting the challenge of predicting rare yet crucial edge cuts. We investigate strategies such as density-based loss weighting and balanced label computation to address distributional imbalance, achieving only modest improvements. Overall, this thesis demonstrates that integrating machine learning into the coarsening phase can significantly enhance multilevel graph partitioning quality.

Zusammenfassung

Die Multilevel Graph-Partitionierung ist eine weit verbreitete Heuristik zur Lösung des $\mathcal{NP}$-complete BALANCED GRAPH PARTITIONING Problems. Sie umfasst drei Phasen: eine Vergröberungs phase, in der die Grösse des Graphen durch das Kontrahieren von Kanten reduziert wird, eine initiale Partitionierungsphase und eine Verfeinerungsphase, in der die Vergröberungsschritte umgekehrt werden, während die Partition verfeinert wird. Die Qualität der finalen Partition hängt von der Effektivität der Vergröberungsphase ab, da eine schlechte Wahl der Kontraktion qualitativ hochwertige Lösungen in den gröberen Graphen zerstören kann. Derzeitige Vergröberungsmethoden, wie z.B. Vertex Clustering, verlassen sich typischerweise nur auf lokale Verbindungen, was ihre Fähigkeit einschränkt, die globalen strukturellen Eigenschaften des Graphen zu erfassen. In dieser Arbeit wird ein neuartiger, auf maschinellem Lernen basierender Vergröberungsalgorithmus vorgeschlagen, der ein tiefes neuronales Netzwerk nutzt, um die Wahrscheinlichkeit vorherzusagen, dass Kanten in qualitativ hochwertigen Partitionen geschnitten werden. Diese Vorhersagen leiten den Clustering-Prozess, wodurch die globale Graphenstruktur berücksichtigt wird. Zusätzlich untersuchen wir Experimente zur Dimensionalitäts- und Merkmalsreduktion, um den Aufwand für die Merkmalsberechnung ohne wesentliche Qualitätsverluste beim Partitionieren zu reduzieren.

Experimente zeigen, dass die geführte Vergröberung die Partitionsqualität erheblich verbessert, insbesondere bei Graphen mit ausgeglichenen Kantenschnitt-Wahrscheinlichkeitsverteilungen. Bei solchen Graphen erreicht unser Ansatz Partitionen mit 2,87% weniger Kantenschnitte als die Standardkonfiguration, was fast dem Gewinn an Partitionsqualität mit perfekter Vorhersage entspricht. Allerdings erreicht die Partitionsqualität nicht ihr volles Potenzial bei Graphen mit stark unausgeglichenen Wahrscheinlichkeitsverteilungen, was die Herausforderung der Vorhersage seltener, aber entscheidender Kantenschnitte hervorhebt. Wir untersuchen Strategien wie dichtebasierte Verlustgewichtung und ausgewogene Label-Berechnung, um Verteilungsungleichgewichte auszugleichen, wobei wir nur bescheidene Verbesserungen erzielen. Insgesamt zeigt diese Arbeit, dass die Integration von maschinellem Lernen in die Vergröberungsphase die Qualität bei der mehrstufigen Partitionierung von Graphen erheblich verbessern kann.

Acknowledgments

An dieser Stelle möchte ich mich herzlich bei allen Personen bedanken, die mich während der Arbeit and dieser Bachelorarbeit unterstützt haben. Mein besonderer Dank gilt meinen Betreuern Nikolai, Daniel und Kenneth, deren fachliche Expertise, wertvolle Anregungen und kontinuierliche Unterstützung wesentlich zum Gelingen dieser Arbeit beigetragen haben. Ebenso danke ich meinen Freunden, die mir immer zur Seite standen und stets für gute Gespräche und Motivation gesorgt haben.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 17. März 2025

Simeon Schrape

Contents

1 Introduction

Graphs are a fundamental mathematical abstraction to represent discrete pairwise relationships between objects. The BALANCED GRAPH PARTITIONING problem, a central issue in graph algorithms, involves dividing a graph into partitions of approximately equal size while minimizing the connections between these partitions. BALANCED GRAPH PARTITIONING has significant practical implications across various domains, addressing complex problems and uncovering structural properties. Some of those applications are load balancing in high-performance computing (HPC) [28], analysis of protein-protein interaction (PPI) networks [29] and VLSI design [21].

Because the BALANCED GRAPH PARTITIONING problem is $\mathcal{NP}$-complete [2], finding an exact solution is computationally infeasible for larger instances. Consequently, approximation algorithms and heuristics are employed to achieve good solutions efficiently. Among these approaches, multilevel graph partitioning has proven particularly effective. This method involves a *coarsening* phase, where the original graph is iteratively simplified by contracting vertices, resulting in progressively smaller graphs. Once a sufficiently small representation is achieved, an *initial partition* is computed. This initial solution is subsequently improved through a *refinement* phase, which systematically reverses the coarsening process, applying local search heuristics to optimize partition boundaries. This multilevel strategy efficiently yields high-quality partitions suitable for practical use in various application domains.

The aim of the coarsening phase is to reduce the graphs size while preserving key structural properties such as clusters and connectivity. To achieve this reduction, highly connected regions must be identified and contracted. Often, this is accomplished using simple heuristics, such as label propagation-based vertex clustering. However, these heuristics rely solely on edge weights, capturing only local connectivity information. This can lead to suboptimal contraction decisions, particularly in highly irregular graphs. Although applying a community detection algorithm as a preprocessing step can improve the clustering, it does not fully overcome this limitation because the initial identification of communities still depends on simple heuristics.

In contrast, this work proposes a more data-driven approach by guiding vertex clustering with a machine learning algorithm. Specifically, a deep neural network is trained on high-quality partitioning data to predict the likelihood that a given edge would be cut. By incorporating these predictions, the proposed coarsening algorithm is able to identify edges that should not be contracted. Consequently, by preserving these critical edges during contraction, the method aims to enhance overall partition quality.

1.1 Contribution

Our core contribution is a coarsening algorithm guided by machine learning, which is discussed in Chapter 4. When clustering the graph before contraction, we use a deep neural network to predict the probability that an edge would be cut in a high-quality final partition. We then use these probabilities to prevent the contraction of edges with high cut probability. This guided clustering integrates additional global structure rather than relying solely on local edge weights. As a result, the quality of the initial partition improves, which in turn leads to a higher-quality final partition. The guided coarsening algorithm is integrated into the Mt-KaHyPar framework and tested against its default configuration.

The experimental evaluation demonstrates the effectiveness of the proposed approach on a set of 69 graph instances, ranging from highly irregular graphs to more structured, engineered systems. The results show a significant reduction in edge cuts compared to the default Mt-KaHyPar configuration. Specifically, on graphs with balanced edge-cut probabilities, the guided coarsening algorithm achieves 2.87% fewer cut edges in the final partition compared to the algorithm without guiding. This reduction is quantified using the geometric mean of the cut edges across all tested graphs. Since the improvement is much smaller on graphs with highly skewed probabilities, we implement density-based loss weighting and label rebalancing to tackle this challenge. However, the results show only modest improvements. Additionally, we investigated dimensionality and feature reduction experiments to reduce feature computation overhead without substantial quality loss. These experiments showed that with only a subset of features, an improvement of 2.72% can still be reached on the subset with balanced distributions.

1.2 Structure of Thesis

In Chapter 2, we introduce the fundamental definitions and concepts used throughout this thesis. We begin with the general graph definitions and the BALANCED GRAPH PARTITIONING problem, then describe deep neural networks for regression, and conclude with a brief overview of principal component analysis. Next, Chapter 3 explores multilevel graph partitioning in more detail, focusing on the Mt-KaHyPar partitioner, which serves as the foundation for our contribution. This Section also provides a concise summary of the tree-structured Parzen estimator for hyperparameter optimization. Our main contribution is discussed in Chapter 4, followed by an experimental evaluation in Chapter 5. Finally, Chapter 6 offers conclusions and suggestions for future work.

2 Fundamentals

In this Section, the fundamental definitions used in this thesis are briefly explained.

2.1 Graphs

The following Sections describe the basic notation of graphs and the GRAPH PARTITIONING problem.

2.1.1 General Definition

An undirected *weighted graph* $G = (V, E, c, \omega)$ is defined as a set of vertices V and a set of edges $E \subseteq \{\{u, v\} \mid u, v \in V\}$. We consider a vertex weight function $c : V \to \mathbb{R}_{>0}$ and an edge weight function $w : E \to \mathbb{R}_{>0}$, both assigning positive weights. The two functions c and ω are extended to sets in a natural way, that means $c(V) := \sum_{v \in V} c(v)$ and $\omega(E) := \sum_{e \in E} \omega(e)$. We define $I(u) := \{\{u, v\} \in E \mid v \in V\}$ as the set of edges where each edge contains u as an endpoint. We also extend it to sets, i.e. $I(C) := \bigcup_{u \in C} I(u)$ where $C \subseteq V$.

2.1.2 Graph Partitioning

The BALANCED GRAPH PARTITIONING problem aims to divide the vertices of a graph into a predefined number of disjoint sets, while minimizing the weight of edges connecting vertices in different sets. A *k-way partition* is defined by a surjective function $\Pi : V \to \{1, \ldots, k\}$, which maps each vertex $v \in V$ to exactly one partition block $V_i = \{v \in V \mid \Pi(v) = i\}$ for $i \in \{1, \ldots, k\}$.

An *ε-balanced k-way partition* is a partition Π, where the total weight of vertices in each partition block V_i does not exceed a maximum block weight $c(V_i) \leq L_{max} := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$, where $\varepsilon \in (0, 1)$ is a balance parameter. The objective is to find a partition Π that minimizes the total weight of the *cut edges*, i.e., edges that connect vertices in different partition blocks. Finding such an optimal partition is an $\mathcal{NP}$-complete problem [2].

2.2 Machine Learning

Machine learning is a field of study that focuses on the development of algorithms capable of learning patterns and insights from data, while trying to generalize their knowledge to previously unseen data. In contrast to traditional algorithms, which are explicitly defined, machine learning algorithms optimize their parameters automatically through a training process. Recent advancements in computational hardware have significantly enhanced the capabilities of machine learning algorithms, particularly in the domain of deep neural networks. These models have demonstrated exceptional performance in solving a wide range of complex problems [6, 13, 20, 30, 32].

The subsequent sections introduce the fundamental concepts of machine learning, with a particular emphasis on deep neural network regression and its underlying principles.

2.2.1 Deep Neural Network Regression

Deep Neural Network (DNN) regression is a supervised machine learning technique in which the model is trained using labeled data. In this context, *labeled data* refers to datasets where both the input features and the corresponding target outputs are provided. The primary objective of the learning process is to derive a mapping from the input space to the output space, that allows it to make accurate predictions on the training data. Additionally the mapping should also make accurate predictions on previously unseen data.

In contrast to the classification tasks, which assign inputs to discrete categorical outputs, regression focuses on the prediction of continuous numerical values. More precisely, given an input vector $\mathbf{x} \in \mathbb{R}^n$, the DNN regression model produces an estimate $\hat{y} \in \mathbb{R}$ through a function defined as

$$\hat{y} = f(\mathbf{x}, \Theta),$$

where f represents the DNN and Θ denotes the set of parameters.

A DNN is a computational model composed of L interconnected layers of artificial neurons, enabling the approximation of complex non-linear functions. The architecture of a DNN for regression tasks can be formally defined as follows.

The input layer receives the feature vector $\mathbf{x} \in \mathbb{R}^n$, where n represents the number of input features.

$$a^0 = \mathbf{x}$$

where a^0 is called the activation vector of the input layer.

The input layer feeds the data into the first hidden layer. Each subsequent hidden layer processes the data further, gradually learning and extracting more complex patterns and relationships, which is the primary function of these layers in a deep neural network. Each hidden layer l, where $l \in \{1, .., L-1\}$ consists of n_l neurons. The activation of each neuron in layer l is computed as an affine transformation followed by a non-linear activation function:

$$z^l = W^l a^{l-1} + b^l$$
$$a^l = \sigma^l(z^l)$$

where $W^l \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix connecting layer $l-1$ to layer l. Each element w_{ij}^l represents the weight of the connection between neuron j in layer $l-1$ and neuron i in layer l. The vector $b^l \in \mathbb{R}^{n_l}$ is the bias vector for the layer l.
The element-wise applied activation function $\sigma^l : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$ introduces nonlinearity, enabling the network to learn complex relationships. There are several activation functions that are commonly used during training of a DNN. The Rectified Linear Unit (ReLU) is defined as

$$\sigma(z) = \max(0, z)$$

Due to its computational efficiency, ReLU is widely used and it helps mitigate the vanishing gradient problem.
The sigmoid function is given by

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

which maps input values to the interval $(0, 1)$ and is particularly suitable for binary classification or probabilities.
The hyperbolic tangent function is expressed as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

producing outputs in the range $(-1, 1)$, thereby centering the data. The leaky ReLU is defined by

$$f(x) = \begin{cases} x, & \text{if } x \geq 0, \\ \alpha x, & \text{if } x < 0, \end{cases}$$

where α is a small constant that permits a non-zero gradient for negative inputs.
The output layer L produces the predicted value $\hat{y} \in \mathbb{R}$. In regression, the output layer typically comprises a single neuron ($n_L = 1$). The output is computed as:

$$\hat{y} = a^L = \sigma^L(W^L a^{L-1} + b^L)$$

where $W^L \in \mathbb{R}^{1 \times n_{L-1}}$ and $b^l \in \mathbb{R}$.
In supervised learning, the learning process involves optimizing the model parameters $\Theta = \{W^1, b^1, \ldots, W^L, b^L\}$ by minimizing a loss function $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$ that quantifies the discrepancy between the predicted $\hat{\boldsymbol{y}} \in \mathbb{R}^N$ and target values $\boldsymbol{y} \in \mathbb{R}^N$. The commonly used loss function for regression that we also use in this thesis is the Mean Squared Error (MSE)

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i=1}^{N} (\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)^2$$

where N is the number of training samples. Backpropagation [31] is the algorithm used to optimize the DNN's parameters. It efficiently computes the gradients of the loss function with respect to each parameter by propagating the error backward through the network. This enables iterative adjustments of weights and biases in the direction of reduced prediction loss, thereby enhancing the model's performance.

2.2.2 Principal Component Analysis

Principal Component Analysis (PCA) is a linear dimensionality reduction technique that projects data onto directions of maximal variance. For a centered data matrix $\tilde{X}$, PCA computes the covariance matrix and obtains its eigen-decomposition:

$$\Sigma = V \Lambda V^\top.$$

By selecting the top k eigenvectors V_k, the data is transformed as

$$Z = \tilde{X} V_k,$$

yielding a lower-dimensional representation that retains the most significant variability.

3 Related Work

In this chapter, we review research that is related our approach. Although our problem is graph partitioning, we utilize a hypergraph partitioning framework (Mt-KaHyPar) for implementation convenience. In this context, each graph edge is treated as a hyperedge connecting two vertices, so the terminology of hypergraph partitioning and graph partitioning can be used interchangeably for our purposes. The discussion is organized as follows: first, we introduce the *Multilevel Graph Partitioning* paradigm by detailing its various phases 3.1. Then we provide an in-depth examination of the Mt-KaHyPar framework 3.2. In Section 3.3, we present the Parzen Tree Estimator, which is used for hyperparameter optimization.

3.1 Multilevel Graph Partitioning

Multilevel graph partitioning (MPG) is a widely adopted method for solving the BALANCED GRAPH PARTITIONING problem. The paradigm was initially introduced by Hendrickson et al. in 1995 [14]. Their findings demonstrated that the application of the multilevel paradigm led to a notable enhancement both in terms of partition quality and reduced computational overhead compared to spectral bisection techniques.

However, the advent of large-scale graph datasets in contemporary applications has presented scalability challenges for traditional sequential partitioning algorithms. In response to these challenges, parallel implementations, such as Mt-KaHyPar (Multi-threaded Karlsruhe Hypergraph Partitioner) [11], Mt-Metis [25], Mt-KaHIP [1], KaMinPar [12] have been engineered to leverage multi-core architectures. These parallel approaches aim to maintain computational scalability.

The MGP paradigm processes input graphs through three stages, as illustrated in Figure 3.1. The *coarsening* phase is characterized by a reduction in graph size, achieved by aggregating vertices into clusters and contracting those clusters into a single coarse vertex. This process yields a hierarchical structure of successively smaller graphs. Upon reaching the most coarse level of this hierarchy, an *initial partition* into k blocks is computed. This initial partition serves as a foundational solution for the subsequent stages of the partitioning process. Following the initial partitioning, the algorithm transitions into the uncoarsening, or *refinement* phase. During refinement, the partitioning solution is projected back up through the hierarchy, and iteratively refined using local search heuristics. This iterative

refinement continues as the algorithm traverses the hierarchy in reverse order, ultimately concluding after the original graph is reconstructed.

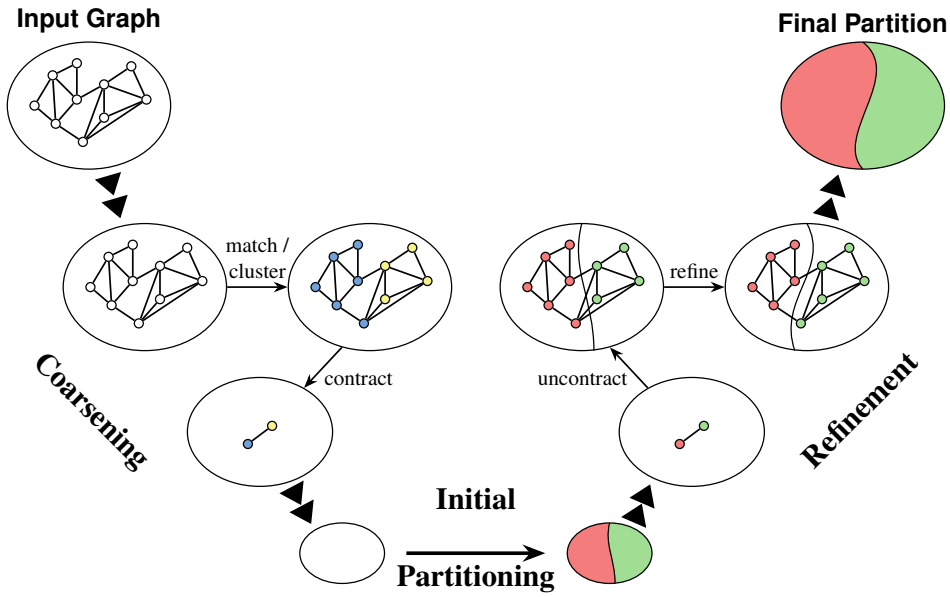

Figure 3.1: Multilevel graph partitioning workflow. The input graph (left) undergoes iterative coarsening to create smaller approximations, followed by initial partitioning on the coarsest graph G_r, and hierarchical refinement during uncoarsening.

3.1.1 Coarsening

In the *coarsening phase* the algorithm creates a sequence of successively smaller (*coarser*) graphs that are structurally similar to each other.

$$\langle G_0 = G, G_1, ..., G_r \rangle$$

Starting at $G_0 = G$, we obtain a coarser graph G_{i+1} in each iteration by contracting some vertices in the finer graph G_i. The goal is to find an initial solution in the coarsest graph G_r that is not significantly worse than a solution that is computed directly on the input graph G. The coarsening algorithm exerts a direct influence on the quality of the initial partitioning. If coarsening iterations are unsuccessful in preserving original structural properties (e.g., clusters, connectivity), and the coarsest graph is, as a result, a poor representation of the input graph, the initial partitioning can lead to larger edge cuts [16, 18]. In order to achieve effective graph coarsening, a precise estimation of which vertices should be contracted to preserve small edge cuts in coarser graphs is necessary. In this context, edges that are between highly-connected vertices are good candidates for contraction. Those highly-connected regions should be found and contracted into a single coarse vertex. The process

of finding those candidates can be achieved by computing a matching or a clustering in the graph. Gottesbüren et al. showed that clustering is particularly effective for reducing the size of graphs with highly skewed node degree distributions, such as social network graphs, and generally produces higher-quality outcomes in the coarsening process [10].

Vertex Clustering

One way to cluster vertices is the *label propagation (LP) based vertex clustering* algorithm. Initially, the algorithm sets every vertex in its own cluster. Subsequently, a random vertex u is picked to select which cluster C the vertex should join. This is achieved by calculating the *heavy-edge* rating function for the vertex and every cluster that shares an edge with the vertex. The cluster with the highest rating is then chosen as the cluster u joins. The rating function is derived from the *heavy-edge* heuristic of Karypis et al. [23] and adapted for plain graphs where each edge is connected to exactly two vertices.

$$r(u, C) := \sum_{e \in I(u) \cap I(C)} w(e)$$

The *heavy-edge* rating function r is defined as the sum of the weights of the edges connecting vertex u to cluster C. Consequently, vertex u is assigned to the cluster C with which it shares the most edges of greatest weight. This approach is consistent with the overall objective of graph partitioning, which is to minimize the weight of cut edges because vertices with a strong connection are more likely to be in the same block of a good partition. Specifically, the contraction of an edge inherently precludes it from being part of the cut set in the initial partitioning phases.

To prevent uncontrolled cluster growth, a constraint on the maximum weight of a cluster, denoted as $c(C)$, is imposed, such that $c(C) \leq c_{max}$. Consequently, if the inclusion of a vertex into a cluster C would cause the cluster weight $c(C)$ to exceed the threshold c_{max}, the vertex is rejected from joining the cluster.

Community aware Vertex Clustering

The *LP-based vertex clustering* is a straightforward heuristic that relies solely on local information, and consequently lacks a global perspective of the graph structure. To address this limitation, Heuer and Schlag [16] introduced a preprocessing step that integrates global information prior to performing vertex clustering. In this approach, community detection algorithms are employed to identify densely connected vertex communities that are only sparsely interconnected. The detection is achieved via modularity maximization using the Louvain method, as originally proposed by Blondel et al. [4]. Once these communities are delineated, the vertex clustering is constrained to operate exclusively within each community, ensuring that vertices from different communities are never clustered together. This method preserves the structural properties derived from the initial community detection phase.

Two-Hop Vertex Clustering

If the clustering and contraction operations fail to meet the required size to compute an initial partition due to weight constraints imposed on the clusters, a procedure known as *two-hop clustering* [12] can be employed. In this approach, a *favored cluster* is assigned to a vertex u when it cannot join any neighboring cluster because of these size constraints. Should the threshold still remain unmet, vertices sharing the same favored cluster are contracted until the target size is achieved.

Contraction

Following the computation of vertex clusters, each cluster undergoes contraction into a single coarse vertex, thereby yielding a reduced graph, denoted as G_{i+1}. The weight of this newly formed coarse vertex is derived as the sum of the vertices' weights within the cluster, represented as

$$\sum_{v \in C} c(v)$$

Edges that exist exclusively within a cluster C are subsequently removed. The coarsening process terminates once the graph is reduced to a predefined number of vertices.

3.1.2 Initial Partitioning

Various approaches are employed by partitioners using the Multilevel Graph Partitioning (MGP) paradigm to implement k-way partitioning during the initial partitioning phase. One common strategy is *recursive bipartitioning*. In this approach, the input graph G_r is first divided into two disjoint vertex sets, $\Pi = \{V_1, V_2\}$. The induced subgraphs $G[V_1]$ and $G[V_2]$ are then recursively partitioned, ultimately yielding a complete k-way partition. To ensure that balance is maintained throughout this process, the ε-constraints are adjusted individually at each bipartitioning step [22].

Alternatively, a k-way partition can be computed directly. One approach to achieve this is *graph growing based partitioning (GGP)* [23]. This method begins by selecting a set of seed vertices, each representing a candidate for a distinct partition. Subsequently, the remaining vertices are iteratively assigned, using a breadth-first search, to the seed with which they share the strongest connection. Since the quality of the partition is strongly influenced by the choice of the initial seeds, the algorithm is executed multiple times, and the partition yielding the best edge cut is chosen.

3.1.3 Refinement

The refinement phase in hypergraph partitioning is intended to improve the quality of the partition obtained during the coarsening and initial partitioning stages. In multi-level

graph partitioning, two commonly used techniques are label propagation and the Fiduccia-Mattheyses (FM) algorithm. The label propagation method, as supported by a variety of practitioners [22, 27], iteratively reassigns boundary vertices to neighboring blocks to maximize local gain while strictly maintaining balance constraints. In contrast, the FM algorithm, originally introduced by Fiduccia et al. [8], systematically identifies vertex moves that produces improvements in partition quality. It achieves this by exploring various move sequences and subsequently reversing any moves that do not contribute to the optimal sequence. As a result, only the most beneficial sequence of vertex moves is applied, thereby further refining the partition quality. Notably, unlike label propagation, the FM algorithm allows moves with negative individual gains, provided that the cumulative gain of the entire move sequence is positive.

Unconstrained Refinement

The two previously discussed refinement strategies are classified as *constrained refinement* since they consider only moves that strictly adhere to the balance constraint. This limitation leads to the omission of moves that, although potentially beneficial, would violate the balance constraint. This issue is addressed by the *unconstrained refinement* paradigm introduced by Maas et al. [26]. In their approach, each refinement step is divided into two phases. In the first phase, unconstrained moves (i.e., moves that may violate the balance constraint) are performed to enhance the quality of the partitioning. In the second phase, a rebalancing algorithm is applied to restore the balance constraint, ensuring that no violations remain. The authors demonstrated that this paradigm achieved a 9.6% reduction in edge cuts on irregular graphs compared to the previous state-of-the-art partitioner.

3.2 Mt-KaHyPar

The Mt-KaHyPar framework is a versatile multilevel graph partitioning framework developed at the Karlsruhe Institute of Technology. It focuses on fast and high quality partitioning solutions by leveraging the use of multi threaded algorithms across all of the three phases (coarsening, initial partitioning, and refinement). This parallel design facilitates the efficient processing of large hypergraphs, which would otherwise present substantial computational challenges for traditional sequential approaches.

3.2.1 Configurations

The Mt-KaHyPar framework supports multiple execution configurations. The default configuration emphasizes speed while still producing high-quality partitioning solutions. It achieves this by integrating parallel algorithms at various stages: parallel community detection as preprocessing, parallel label propagation-based vertex clustering for coarsen-

ing, parallel recursive bipartitioning for the initial partitioning, and a fully parallel k-way FM gain exchange for refinement. Although the extensive use of parallel algorithms may slightly compromise solution quality, this configuration outperforms nearly all existing hypergraph partitioners in both partitioning quality and execution time [10].

In contrast, the quality configuration prioritizes solution quality at the expense of runtime. In this mode, flow-based refinement methods are employed to significantly enhance partition quality, however, this is accomplished at the expense of reduced running time. Moreover, a high-quality mode can be activated, which integrates an n-level partitioning algorithm that employs a fine-grained coarsening strategy, i.e. contracting the graph by only a single vertex at each step. Although this deep hierarchical approach improves quality by merely 0.5% compared to the standard quality configuration, it nearly doubles the execution time. Nonetheless, the high-quality results remain competitive with those produced by the best sequential multilevel graph partitioner, KaHyPar [1], while still offering a faster overall performance.

The final configuration is the deterministic mode, designed to ensure reproducible results when using the same input graph and seed. This mode eliminates run-to-run variations, usually induced by concurrency, by employing deterministic implementations for the three partitioning phases [9].

3.2.2 Additional Features

Beyond its configurable execution modes, Mt-KaHyPar distinguishes itself through several additional advanced features. First, the partitioner is equipped with optimized data structures tailored for plain graphs. When operating on plain graphs rather than hypergraphs, it internally switches to a more efficient data structure, resulting in an approximate factor 2 speedup and establishing Mt-KaHyPar as a state-of-the-art solver for graph partitioning [10].

Another notable feature is the support for fixed vertices. Mt-KaHyPar allows the pinning of selected vertices to predefined blocks, a functionality that is particularly useful when partitioning solutions must incorporate pre-located components, as is often required in real applications.

Furthermore, the partitioner offers the capability to specify individual block weights. This feature allows users to assign distinct weights to the k partitioning blocks, thereby effectively disabling the standard balance constraint. This flexibility is especially advantageous in load balancing scenarios where components differ in work capacity, as it enables the workload to be distributed in accordance with the predefined block capacities.

[1] `https://kahypar.org/`

3.2.3 Objective Functions

In addition to these features, Mt-KaHyPar supports multiple objective functions. The two most commonly used metrics are the cut-net metric and the connectivity metric. The cut-net metric is defined as the sum of the weights of all nets that span multiple blocks:

$$f_c(\Pi) \coloneqq \sum_{e \in E_{\text{Cut}}(\Pi)} \omega(e).$$

The connectivity metric extends this by weighting each net according to the number of blocks it connects minus one:

$$f_{\lambda-1}(\Pi) \coloneqq \sum_{e \in E_{\text{Cut}}(\Pi)} (\lambda(e) - 1) \cdot \omega(e).$$

A third metric, the sum of external degrees, is equivalent to the connectivity metric without the subtraction, i.e.,

$$f_s(\Pi) \coloneqq \sum_{e \in E_{\text{Cut}}(\Pi)} \lambda(e) \cdot \omega(e).$$

In these formulas, $E_{\text{Cut}}(\Pi)$ denotes the set of nets that span across blocks, and $\lambda(e)$ represents the number of blocks connected by net e.

Moreover, Mt-KaHyPar supports the Steiner tree metric, which transforms the partitioning problem into a mapping problem. In this configuration, the user supplies an additional target graph G, where nodes represent partitioning blocks and edge weights define the cost of inter-block connections. The partitioner then maps the hypernodes onto G, aiming to minimize the sum of the Steiner trees[2] induced by the nets of the input hypergraph. The Steiner tree metric is formulated as:

$$f_{ST} \coloneqq \sum_{e \in E} \text{DIST}_G(\Lambda(e)) \cdot \omega(e),$$

where $\text{DIST}_G(\Lambda(e))$ is the weight of the minimal Steiner tree connecting the blocks $\Lambda(e)$ spanned by net e. This metric is particularly beneficial in modeling wire length in VLSI design [15].

In summary, Mt-KaHyPar is a versatile and powerful tool, well-suited to a range of partitioning problems through its diverse set of features and objective functions.

3.3 Tree-structured Parzen Estimator

Sequential Model-Based Optimization (SMBO) provides an effective framework for optimizing expensive black-box functions. Within this context, the Tree-structured Parzen

[2] https://en.wikipedia.org/wiki/Steiner_tree_problem#Steiner_tree_in_graphs_and_variants

Estimator (TPE), introduced by Bergstra et al. [3], has become a prominent algorithm for hyperparameter optimization in machine learning.

TPE diverges from traditional Gaussian Process-based Bayesian Optimization by modeling the search distribution indirectly. Instead of approximating the objective function directly, TPE estimates two probability densities via kernel density estimation (KDE): one, $l(x)$, for hyperparameter configurations that yield objective values better than a predetermined threshold, and another, $g(x)$, for those that do not. The ratio $l(x)/g(x)$ then serves as an acquisition function, effectively guiding the search towards regions in the hyperparameter space that are statistically more likely to contain optimal configurations.

Empirical evaluations demonstrate that TPE is particularly efficient in high-dimensional settings and is capable of handling both continuous and discrete hyperparameters [3]. Its non-parametric approach, which reduces assumptions about the underlying objective function, along with its computational efficiency, has solidified TPEs status as a robust and influential method in automated hyperparameter optimization.

4 Guided Coarsening using Machine-Learning

Section 4.1 provides a broad overview and motivation for our guided coarsening approach. Following this, we detail the feature selection (4.2) and feature computation (4.3) processes. Next, Section 4.4 explains the label computation for our supervised learning process. We then discuss the complete training process in 4.5. Finally, we conclude this chapter with a detailed explanation of guided vertex clustering in Section 4.6.

The feature selection, feature computation, label computation, and implementation of the guided vertex clustering in the MtKaHyPar framework were performed by my supervisor Nikolai Maas, as these tasks were beyond the intended scope of a Bachelor Thesis.

4.1 Overview

As stated before, the coarsening phase is a pivotal component of multilevel partitioning, as the quality of the coarsening algorithm directly influences the quality of the coarsest graph on which the initial partitioning is performed. An inadequate approximation of the structural properties from one coarsening level to the next often results in a suboptimal initial partition that even subsequent refinement procedures cannot fully remedy. Although heuristic methods such as vertex clustering, as implemented in Mt-KaHyPar, offer simplicity and computational efficiency, they are inherently limited by their exclusive reliance on local connectivity information [16]. This limitation motivates the exploration of more sophisticated clustering strategies that integrate a broader range of structural features, with the aim of enhancing both the quality of the coarsened representations and the final partitioning outcome. As discussed in Section 3.1.1, we saw that community detection can incorporate global structure before clustering. Our approach takes this idea further by using a machine learning model to predict edge cut probabilities, effectively learning global structural cues from data rather than relying on a pre-clustering of the graph. This guided coarsening can adapt to patterns that simple heuristics might miss.
In particular, we define a machine learning model

$$f_{\mathcal{M}} : E \to [0, 1],$$

which assigns each edge a probability in the interval $[0, 1]$, quantifying the likelihood that the edge would be cut in a high-quality final partition. This probabilistic assessment guides

the coarsening phase via a mechanism we term *guided vertex clustering*. In this approach, computed probabilities filter candidate clusters a vertex might join. Specifically, since high-probability edges should remain uncontracted, clusters with an average probability above a predefined threshold are excluded. If the resulting contraction is insufficient, the threshold is increased.

4.2 Feature Selection

Our machine learning model uses a total of 208 features divided into three categories: *global graph features*, *edge-specific features*, and *vertex-specific features*. The *global graph features* describe overarching properties of the graph (e.g., the number of vertices and edges). These form the first 28 components of the feature vector:

$$\mathbf{x}_{\mathrm{g}} = (x_1, x_2, \ldots, x_{28}).$$

Next, the *edge-specific features* capture pairwise relationships between vertices (e.g., locality, jaccard index, and other similarity metrics). These 70 features make up

$$\mathbf{x}_{\mathrm{e}} = (x_{29}, x_{30}, \ldots, x_{98}).$$

Finally, the *vertex-specific features* capture individual attributes of the two endpoints u and v. Specifically, 55 features are calculated for each vertex, giving us

$$\mathbf{x}_u = (x_{99}, \ldots, x_{153}) \quad \text{and} \quad \mathbf{x}_v = (x_{154}, \ldots, x_{208}),$$

covering characteristics such as vertex degrees and neighbor information.

Concatenating these four blocks results in the complete 208-dimensional feature vector

$$\mathbf{x} = (x_1, x_2, \ldots, x_{208}) \in \mathbb{R}^{208}.$$

We compute one such feature vector x for every edge $\{u, v\} \in E$.

4.3 Feature Computation

Some features incur significant computational costs. In the absence of substantial feature reduction, it is infeasible to directly implement the model predictions into the partitioning algorithm without a severe running time penalty. Consequently, the evaluation workflow for this thesis is as follows. All features are precomputed and stored alongside their corresponding structural graph data. Subsequently, these features are used to predict the probability values for the entire graph, which are then also stored. Finally, for the partition quality evaluation of the guided Mt-KaHyPar algorithm, these precomputed probability files, in conjunction with the graph data, are provided as input to the algorithm. The complete workflow is depicted in Figure 4.1.

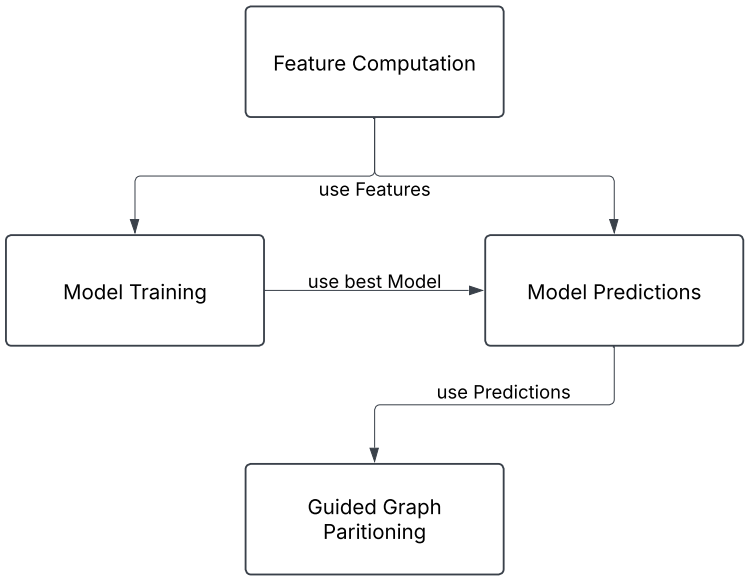

Figure 4.1: Workflow of this Thesis

4.4 Label Computation

In the context of this thesis, label computation holds a crucial role in the overall methodology. As established in the preceding sections, the labels are designed to represent the probability of an edge being cut within a high-quality partition. To derive these probabilistic labels, we employ a high-quality graph partitioning algorithm, which is applied iteratively to each graph instance in the dataset. For each execution of the partitioning algorithm, we record, for every edge in the graph, whether it is cut or uncut in the resulting partition. Aggregating the results across multiple partitioning runs yields a frequency count for each edge, indicating the number of times it was identified as a cut edge. To enhance the diversity of the generated labels and to obtain more robust frequency estimates, we systematically vary the parameter k across the partitioning runs. This approach ensures that the trained model is exposed to and can effectively generalize across a spectrum of partitions. Ultimately, the probability label for each edge is computed by dividing its frequency count by the total number of partitioning executions, thus providing an empirical estimate of the edge cut probability.

4.5 Model Training

The machine learning model $\mathcal{M}$ that is used throughout this thesis is a deep neural network with a regression output as it was introduced in Section 2.2.1. It is trained with the mean squared error loss function.

Since the objective of this thesis is to predict values within the range $[0, 1]$, the sigmoid activation function is chosen for the output layer, due to its inherent property of constraining output values to this interval, such that $\hat{y} \in [0, 1]$.

We train our DNN using mini-batches over multiple epochs. In each epoch, the model processes the entire training set in chunks of size κ. This approach combines the advantages of batch gradient descent (BGD), which updates parameters only after the entire dataset has been processed, with stochastic gradient descent (SGD), which updates parameters after every individual sample. By updating parameters on each mini-batch, we strike a balance between BGDs comprehensive gradient calculation and SGDs more frequent updates. The number of epochs τ and the mini-batch size κ are determined through hyperparameter optimization.

The neural network architecture is given by the tuple

$$\mathcal{A} = (n_0, n_1, \ldots, n_L).$$

Because the input contains 208 features, the first layer has $n_0 = 208$ neurons. Our goal is to predict a continuous value, so the final layer has exactly one neuron, $n_L = 1$. The layers in between, $n_1, \ldots, n_{L-1}$, are the hidden layers. Both the number of these layers and the number of neurons in each layer are chosen during hyperparameter optimization.

For improvement of the learning abilities of the DNN we use the Adam optimizer [24]. Adam improves learning by using momentum to escape local minima.

The learning rate, denoted as η, is a crucial hyperparameter that controls the step size during the parameter update process in training. A large learning rate facilitates rapid convergence but risks overshooting the optimal solution, leading to oscillations. Conversely, a small learning rate ensures more stable convergence but can be slow and increase the likelihood of becoming trapped in local optima, even when employing adaptive optimizers like Adam. To mitigate these trade-offs and promote both fast and stable convergence, we implemented two strategies.

The first strategy involves employing a learning rate scheduler, which dynamically adjusts the learning rate based on the validation set performance. Specifically, the scheduler implemented in this work monitors the validation loss and reduces the learning rate by a factor of 10 when a plateau is detected.

The second strategy focuses on stabilizing the training process through batch normalization, an optional feature controlled by the hyperparameter $\beta \in \{0, 1\}$ to enable or disable it. Batch normalization, initially proposed by Ioffe and Szegedy [17], is a technique designed to alleviate internal covariate shift, thereby accelerating training and improving network

stability. This is achieved by integrating batch normalization layers into the model architecture. In particular, a batch normalization layer is inserted after each neuronal layer and before the activation function. This layer normalizes the input batches, reducing the distributional shift of layer activations, which allows for the use of higher learning rates therefore contributes to faster convergence.

With all hyperparameters together we define our model as the tuple

$$\mathcal{M} = (\tau,\, \kappa,\, a,\, \mathcal{A},\, \eta,\, \beta,\, \lambda,\, \rho,\, \delta)$$

Where ρ, δ, λ are regularization hyperparameters which are explained in Section 4.5.3 and a is the data amount as described in Section 4.5.1

4.5.1 Training Data

The training data for the model is derived from the precomputed graph features. We define the training set as a collection of edges, that stem form a set of different graphs, and construct the feature matrix by concatenating the features from each graph. To control the amount of data, we introduce the hyperparameter a. Due to the high computational cost associated with training a neural network, it is impractical to utilize all edges from every graph. Therefore, we specify a fixed total number of data points and allocate this quantity evenly across the graphs in the training set. From each graph, this number of edges is randomly sampled and added to the feature matrix. Consequently, each graph is represented by an equal number of edges, regardless of variations in graph size or edge count.

4.5.2 Input Scaling

Proper input scaling is a crucial preprocessing step in neural network training. We employ z-score normalization to transform each feature x_i into

$$\hat{x} = \frac{x - \mu}{\sigma},$$

where μ and σ are the mean and standard deviation computed from the training data. This transformation ensures that every feature has a mean of 0 and a standard deviation of 1.

After fitting the scaler on the training set, the same parameters are used to normalize the evaluation set, ensuring consistency between both datasets. This approach improves weight initialization and facilitates smoother convergence during optimization. Moreover, it prevents features with larger numerical ranges from dominating the loss function, thereby allowing the network to learn balanced representations from all input features.

4.5.3 Overfitting

Overfitting is a prevalent challenge in training deep neural networks. When a model overfits, it learns patterns that are overly specific to the training data and consequently fails to generalize to new, unseen data. To mitigate this issue, we implemented three regularization techniques: *early stopping*, *dropout*, and *weight decay*.

Early Stopping

Early stopping is a regularization strategy designed to prevent overfitting by terminating the training process before the model begins to fit noise in the data. This method employs a patience hyperparameter, denoted by ρ, to monitor the model's performance on a validation set. After each epoch, the model's performance is evaluated; if no improvement is observed relative to the best performance recorded, a counter is incremented. Once this counter reaches the value specified by ρ, training is halted, and the model state corresponding to the best validation performance is selected as the final model.

Dropout

The dropout technique is applied at the architectural level of the neural network. In this method, a dropout layer is inserted after every layer except for the final layer, and it contains the same number of neurons as the preceding layer. In the dropout layer, each neuron is connected to exactly one neuron from the preceding layer and is fully connected to the subsequent layer. A hyperparameter δ,in the range $[0, 0.5]$, is introduced to specify the probability that a neuron's output is set to zero, effectively blocking the activation of the corresponding neuron in the preceding layer. Neurons that are not dropped retain their activation, thereby simply passing the signal forward. As a result, during training, the model repeatedly ignores random subsets of neurons. This process can be interpreted as simultaneously training an ensemble of sub-networks, which encourages the model to learn robust mappings and connections. Consequently, the dropout technique enhances the model's ability to generalize by reducing its reliance on any single neuron.

Weight Decay

Weight decay is a regularization technique that incorporates an L2-norm penalty on the network weights into the loss function. Formally, the modified loss function is given by

$$L_\lambda(y, \hat{y}) = L(y, \hat{y}) + \lambda \sum_{i=1}^{L} \|W^i\|^2,$$

where $L(y, \hat{y})$ denotes the MSE loss, W^i is the weight matrix of the i-th layer (as defined in Section 2.2.1), and λ is a hyperparameter controlling the regularization strength. This

penalty term imposes a higher cost on larger weights, thereby encouraging the optimization process to favor smaller weight values. Consequently, the model is less prone to developing overly complex representations, which enhances its generalization performance and mitigates the risk of overfitting.

4.5.4 Train-Validation Split

To effectively train our deep neural network, we partition the input data into two disjoint sets: one for training and one for validation. The ratio of the split is determined by the overall data volume which itself is controlled by the hyperparameter a. In this study, 80% of the data is allocated to the training set, while the remaining 20% is reserved for validation. The training set is employed to optimize the network parameters, whereas the validation set is used for early stopping and hyperparameter tuning. For the final evaluation, predictions of all edges from a single graph are generated and subsequently used to perform guided coarsening. Although there is a minor overlap between the final evaluation data and the training data, stemming from the use of the same graph set, the comparatively smaller size of the training data justifies the assumption that this strategy yields valuable insights into the model's ability to enhance partition quality during coarsening. Partition quality is benchmarked against both the guided Mt-KaHyPar algorithm, which incorporates ground truth values, and the Mt-KaHyPar algorithm using the default configuration.

4.5.5 Hyperparameter Optimization

Hyperparameter optimization plays a crucial role in training deep neural networks by systematically tuning the parameters that govern the learning process. Common approaches include grid search and random search, which explore hyperparameter combinations either through all possible combinations or via random sampling, though both can be computationally inefficient for large spaces.

Bayesian optimization offers a more efficient alternative by constructing a probabilistic surrogate model of the objective function. Techniques such as Gaussian processes [19] or the Tree-structured Parzen Estimator (TPE) 3.3 predict the performance of different hyperparameter configurations and guide the search toward the most promising regions of the space. This iterative process minimizes the number of expensive function evaluations, making it particularly suitable when training trials are computationally costly.

For our approach, we define a search space for each tunable hyperparameter, which may be categorical or numerical. To improve computational costs, we fix the dataset size for optimization trials to the predetermined value a which is set to a relatively small number. Each trial trains a complete model on this smaller subset, and the optimal hyperparameter configuration is subsequently applied to a larger training set. Additionally, we employ a median pruner that terminates trials if their best intermediate performance is below the median of previous trials at the same stage. The TPE algorithm then focuses on minimizing

the average mean squared error on the validation set that was recorded during the best epoch of training.

4.6 Guided Vertex Clustering

To enhance the vertex clustering process within Mt-KaHyPar as it was introduced in Section 3.1.1, a guided vertex clustering approach is implemented. This method refines the cluster assignment by incorporating edge probabilities and a dynamic threshold. When considering a vertex u for clustering, the algorithm evaluates potential clusters C based not only on the sum of edge weights connecting u to C, but also on the average probability of these connecting edges.

A guiding threshold t is introduced, initially set to 0.2. For each potential cluster C, the average probability $\bar{p}(u, C)$ of edges connecting u to C is computed as:

$$\bar{p}(u, C) = \frac{1}{r(u, C)} \sum_{e \in E_{u,C}} f_{\mathcal{M}}(e) \cdot \omega(e)$$

where $E_{u,C} \coloneqq I(u) \cap I(C)$ is the set of edges between vertex u and cluster C, $f_{\mathcal{M}}(e)$ is the probability of edge e, $\omega(e)$ is the weight of an edge, and $r(u, C)$ is the heavy edge rating function and therefore the sum of the weights.

If $\bar{p}(u, C) > t$, cluster C is excluded as a candidate for u to join. This exclusion is intended to guide the clustering process away from forming clusters connected by edges with high probability of being cut in the final partition.

For clusters C that are not excluded (i.e. $\bar{p}(u, C) \leq t$) the algorithm further scales the rating of the clusters to influence the clustering decision. The potential clusters C that u might join are assigned a rating r_{new} based on a quadratic scaling function:

$$s = \frac{\max(0, t - \bar{p}(u, C))}{t}$$

$$r_{new} = s^2 \cdot r(u, C)$$

This quadratic scaling reduces the effective edge rating more significantly for connections where the average probability $\bar{p}(u, C)$ is closer to the threshold t, further discouraging joining such clusters.

During the coarsening contraction, new edges are formed which might be an aggregate of multiple edges from the previous coarsening level. The probability for these newly formed edges is computed as a weighted average of the probabilities of the original edges that are contracted. Specifically, for a new edge e_{new} formed by contracting clusters C_1 and C_2, its probability $f(e_{new})$ is calculated as:

$$f_{\mathcal{M}}(e_{new}) = \frac{\sum_{e \in E_{C_1,C_2}} f_{\mathcal{M}}(e) \cdot \omega(e)}{\sum_{e \in E_{C_1,C_2}} \omega(e)}$$

where E_{C_1,C_2} is the set of original edges between clusters C_1 and C_2, $f_{\mathcal{M}}(e)$ is the probability of an original edge e, and $\omega(e)$ is its weight. This weighted average ensures that the probabilities of newly formed edges in the coarser hypergraph meaningfully reflect the probabilities of the original edges, giving more influence to probabilities associated with heavier edges.

To manage the constraints imposed by the threshold, the algorithm employs a series of guided subrounds. If a coarsening round does not achieve sufficient contraction due to cluster exclusions, the threshold t is increased stepwise from 0.2 to 0.8 in increments of 0.2. This gradual increase in the threshold relaxes the guiding constraints over subsequent subrounds, allowing for more contractions to occur. The total number of guided subrounds is controlled by a parameter, and set to 5 in the current configuration, resulting in 4 guided subrounds with increasing thresholds after the initial clustering round.

5 Experimental Evaluation

This Section presents the evaluation of the guided coarsening algorithm's partition quality. We begin by outlining the experimental setup in Section 5.1, then analyze the resulting partition quality in Section 5.2.

5.1 Experimental Setup

The experimental setup Section discusses the implementation (Section 5.1.1), the evaluation environment (Section 5.1.2), the test instances (Section 5.1.3), and the performance profiles (Section 5.1.4) used to graphically analyze the results.

5.1.1 Implementation

All training and model prediction code is implemented in PYTHON 3.13. For training, we employed two machine learning frameworks: PYTORCH[1] and SCIKIT-LEARN[2]. Data handling was performed using the PANDAS library[3]. Hyperparameter optimization was conducted with OPTUNA[4], and the training process was monitored via WEIGHTS AND BIASES[5].

The guided coarsening code is integrated into the Mt-KaHyPar partitioner framework (version 1.4)[6]. Implementation of the code is done in C++. It is compiled with the GCC compiler (version 14.2.0).

5.1.2 Evaluation Environment

For each model evaluation, we execute our guided coarsening partitioner on all test graphs under two different configurations. Specifically, one variant employs unconstrained refinement (see Section 3.1.3), while the other applies the constrained refinement scheme. We

[1] https://pytorch.org/
[2] https://scikit-learn.org/
[3] https://pandas.pydata.org/
[4] https://optuna.org/
[5] https://wandb.ai/
[6] https://github.com/kahypar/mt-kahypar/tree/nikolai/guided-coarsening

adopt the cut-net metric function (see Section 3.2.3) as the objective and allow a 3% imbalance in the partitions. The cut-net metric on plain graphs corresponds to the total number of cut edges in the final partition. Each partitioner is evaluated for four distinct values of k (namely $k = 4, 16, 32, 64$) and executed with 64 threads. Furthermore, each configuration is repeated five times to obtain robust average results.

As our evaluation prioritized enhancing partition quality over execution time, we employed a heterogeneous set of hardware. All machines ran on LINUX KERNEL version 5.14.0 with the ROCKY LINUX 9.5 (BLUE ONYX) operating system. The hardware configurations varied in terms of CPU and RAM. The least powerful machine featured four INTEL® XEON® GOLD 6138 processors with a clock speed of 2.0 GHz and 754 GiB of memory, while the most powerful machine was equipped with an AMD EPYC 9684X 96-Core Processor running at 2.55 GHz and 1.5 TiB of memory.

5.1.3 Instances

The graph set used throughout this thesis consists of 69 graphs that capture a broad spectrum of network structures and complexities. These instances were originally compiled by Maas et al. [26]. They have been selected to ensure they represent both naturally occurring and engineered networks. In our analysis, the graphs are characterized along two primary dimensions: structural regularity and data origin. A subset of these graphs is irregular, exhibiting heterogeneous connectivity typical of real-world networks such as social, citation, or web graphs. In these irregular instances, certain nodes have many connections while others remain sparsely linked. Conversely, another subset displays a more homogeneous, grid-like structure, as commonly seen in engineered networks like road maps or circuit layouts. Additionally, the dataset includes synthetic models generated by algorithms such as R-MAT, RHG, Delaunay, and random geometric methods. The range of graphs ensures that our proposed method is thoroughly tested under a variety of real-world and controlled conditions. As described in Section 4.5.4, the same set of graphs is used for both training and final evaluation. However, only a small subset of edges from each graph is used during training, whereas all edges are used in the final evaluation.

5.1.4 Performance Profiles

Performance profiles provide a systematic framework for comparing the effectiveness of various graph partitioning algorithms across a set of benchmark instances. Originally introduced by Dolan and Moré [7], these profiles offer a quantitative method for assessing the relative performance of competing algorithms.

Let $\mathcal{S}$ denote the set of solvers under evaluation. In our study, this set comprises the guided coarsening algorithms that employ different machine learning models to predict probabilities, the guided coarsening algorithm that utilizes the ground truth probabilities, and the Mt-KaHyPar algorithm in default configuration without guided coarsening. Furthermore,

let $\mathcal{P}$ be the set of benchmark problems, where $|\mathcal{P}|$ is the total number of instances. Each instance consists of a specific graph and a corresponding k value. As explained in Section 5.1.2, we run each instance five times and use the arithmetic mean of these runs to obtain reliable results.

For each algorithm $s \in \mathcal{S}$ and each instance $p \in \mathcal{P}$, we define the performance ratio as

$$r_{s,p} = \frac{f\left(\Pi_{s,p}\right)}{\min\{f\left(\Pi_{s,p}\right) : s \in \mathcal{S}\}},$$

where $\Pi_{s,p}$ is the partition produced by algorithm s on instance p and $f\left(\Pi_{s,p}\right)$ denotes the cut metric evaluated for that partition. In this thesis, we assess the quality of the cut at two distinct stages of the algorithm. The first evaluation is carried out after the first refinement step, and we refer to it as the **Level 1 Cut** (LV1C). The second evaluation is performed on the fully refined final partition, which we denote as the **Final Cut** (FC). A performance ratio of $r_{s,p} = 1$ indicates that algorithm s achieves the best performance on instance p, while values greater than 1 indicate the degree to which the performance of s deviates from the optimum.

The performance profile of algorithm s is then defined as the cumulative distribution function of the performance ratios:

$$\rho_s(\tau) = \frac{1}{|\mathcal{P}|} \left|\{p \in \mathcal{P} \mid r_{s,p} \leq \tau\}\right|, \quad \tau \in \mathbb{R}.$$

This function $\rho_s(\tau)$ represents the proportion of instances for which the performance of algorithm s is within a factor τ of the best performance. In particular, $\rho_s(1)$ indicates the fraction of instances in which algorithm s is the best-performing method.

In the context of this thesis, performance profiles serve as a critical tool for evaluating the impact of the proposed guided coarsening approach. They provide a robust means of comparing the extent to which the guided coarsening strategy enhances partition quality relative to conventional methods.

5.2 Partition Quality Evaluation

In this section, we first present our initial hypothesis (Section 5.2.1) and then further investigate the behavior in Section 5.2.2. We subsequently address the issue of unbalanced distributions (Section 5.2.3). Finally, we provide a brief evaluation of principal component analysis and feature reduction in Section 5.2.4

5.2.1 Initial Hypothesis

We initially hypothesized that the partition quality could be enhanced by employing guided coarsening across all graphs. To test this hypothesis, we trained a deep neural network on a limited set of edges extracted from each graph, as described in Section 4.5.1. Subsequently, we evaluated the partition quality using performance profile plots generated from a test set of 20 graphs, randomly sampled from the 69 graphs used for training.

The models hyperparameters were determined via the Tree-structured Parzen Estimator (TPE) optimization process comprising 300 trials with trial pruning activated, as detailed in Section 4.5.5. For the optimization phase, we utilized a dataset of $a = 138,000$ samples, equivalent to 20,000 samples per graph. The optimization yielded an architecture with three hidden layers $\mathcal{A} = (208, 1549, 2045, 702, 1)$. A dropout rate of $\delta = 0.2$ was applied, while batch normalization was disabled ($\beta = 0$). The network was trained for $\tau = 223$ epochs with a batch size of $\kappa = 512$ and an early stopping patience parameter of $\rho = 38$. The initial learning rate was set to $\eta = 0.0004$, and a weight decay rate of $\lambda = 0.000004$ was employed.

After optimization, the final model was trained on a larger dataset of $a = 1\,000\,000$ samples using the hyperparameters obtained from the TPE process. We formally define the models hyperparameters as the tuple

$$\mathcal{M}_1 = (\tau,\, \kappa,\, a,\, \mathcal{A},\, \eta,\, \beta,\, \lambda,\, \rho,\, \delta)$$

The partition quality is shown in Figure 5.1. Although the setup yielded an improvement over the standard Mt-KaHyPar partitioner without guided coarsening (denoted as MT-KAHYPAR), its partition quality still lagged behind that of the guided variant using ground truth labels (MT-KAHYPAR-GTL, where GTL stands for **G**uided with **T**raining **L**abels). As shown in Table 5.1, the algorithm employing guided coarsening with model $\mathcal{M}_1$ achieved a 1.16% reduction in edge cuts compared to the unguided version, yet incurred 3.11% more cuts than the algorithm guided by the training labels. The edge cut values are computed by first taking the arithmetic mean over the five runs, then calculating the geometric mean over the four distinct k-values, and finally obtaining the overall geometric mean across all graphs. We therefore conclude that the model significantly enhances partition quality; however, it does not fully realize its potential.

This phenomenon is likely attributable to the highly skewed distribution of the ground truth probabilities, as illustrated in Figure 5.2.

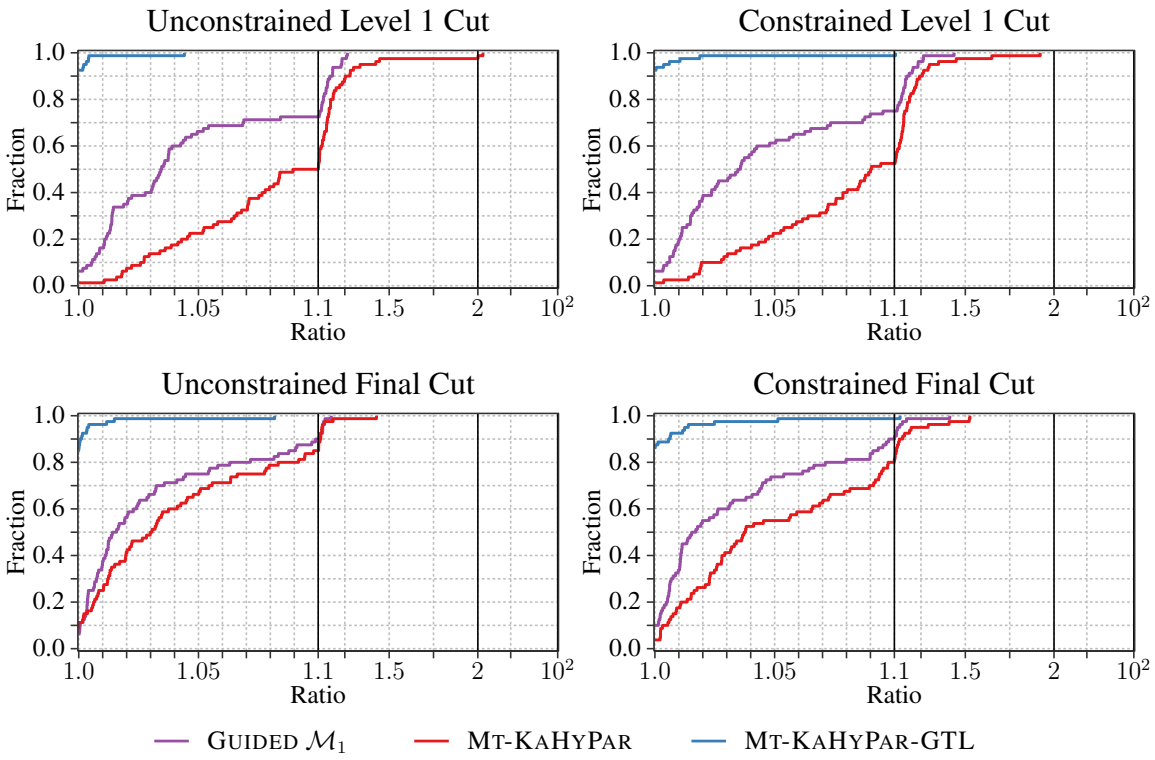

Figure 5.1: Partition quality of Model 1 on a test set of 20 graphs. The left figures show results using strong, unconstrained refinement, with cut size measured after one refinement step and the cut size in the final partition. The right figures present outcomes for the partitioner using weaker, constrained refinement, evaluated on the same two stages in the algorithm.

Refinement	Model	Reference Model	Level 1 Cut	Final Cut
UC	GUIDED $\mathcal{M}_1$	MT-KAHYPAR	-6.77%	-1.16%
C	GUIDED $\mathcal{M}_1$	MT-KAHYPAR	-5.69%	-2.86%
UC	GUIDED $\mathcal{M}_1$	MT-KAHYPAR-GTL	+5.72%	+3.11%
C	GUIDED $\mathcal{M}_1$	MT-KAHYPAR-GTL	+5.99%	+3.42%

Table 5.1: Difference of the edge cut amount (geometric mean) of the GUIDED $\mathcal{M}_1$ algorithm in comparison to the unguided MT-KAHYPAR algorithm and the training labels guided MT-KAHYPAR-GTL algorithm. Both with constrained (C) and unconstrained (UC) refinement.

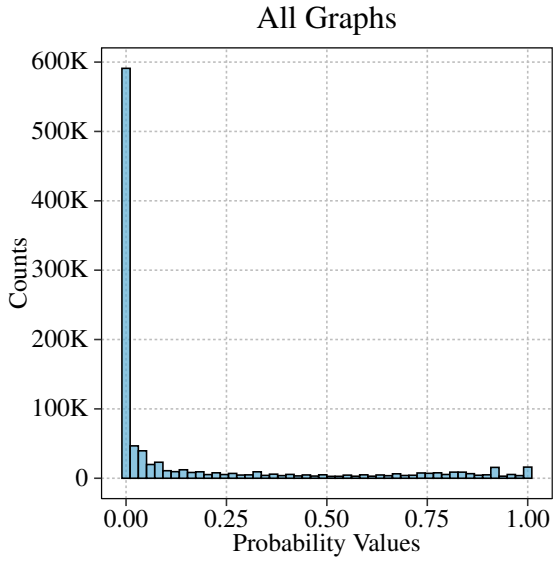

Figure 5.2: Distribution of ground truth probabilities: The data was sampled using the same method as the training data, with a total of 1 000 000 samples.

Consequently, the model tends to overpredict probabilities near zero. Although the overall mean squared error remains low, the model exhibits a high error rate for values approaching one, since such errors occur much less compared to mispredictions of low values. In the context of the guiding algorithm, this behavior allows the contraction of edges that possess a high probability of being cut in a high-quality final partition. As a result, the initial partitioning deteriorates because it cannot cut those previously contracted edges that would have contributed to an improved final partition.

5.2.2 Further Investigation

A further question arises as to whether the model exhibits these issues uniformly across all graphs or if they are confined to specific subsets. To explore this, we computed the mean μ of the probability values for each graph in the dataset, using it as an indicator of distribution skewness; a lower mean implies a higher concentration of probabilities near zero. We then sorted the graphs by their respective mean values and partitioned the dataset into three roughly equal sized subsets. The first subset comprises the 20 graphs with the highest mean values, the second contains graphs 21 to 45, and the third includes graphs 46 to 69 (Figure 5.5). Examination of the label distributions for these subsets (see Figure 5.4) reveals that the first subset exhibits a considerably less skewed distribution compared to the other two.

Finally, we evaluated model $\mathcal{M}_1$ on each of the three subsets (see Figure 5.3). The evaluation on the first 20 graphs (Table 5.2) illustrates that in the unconstrained setting, model

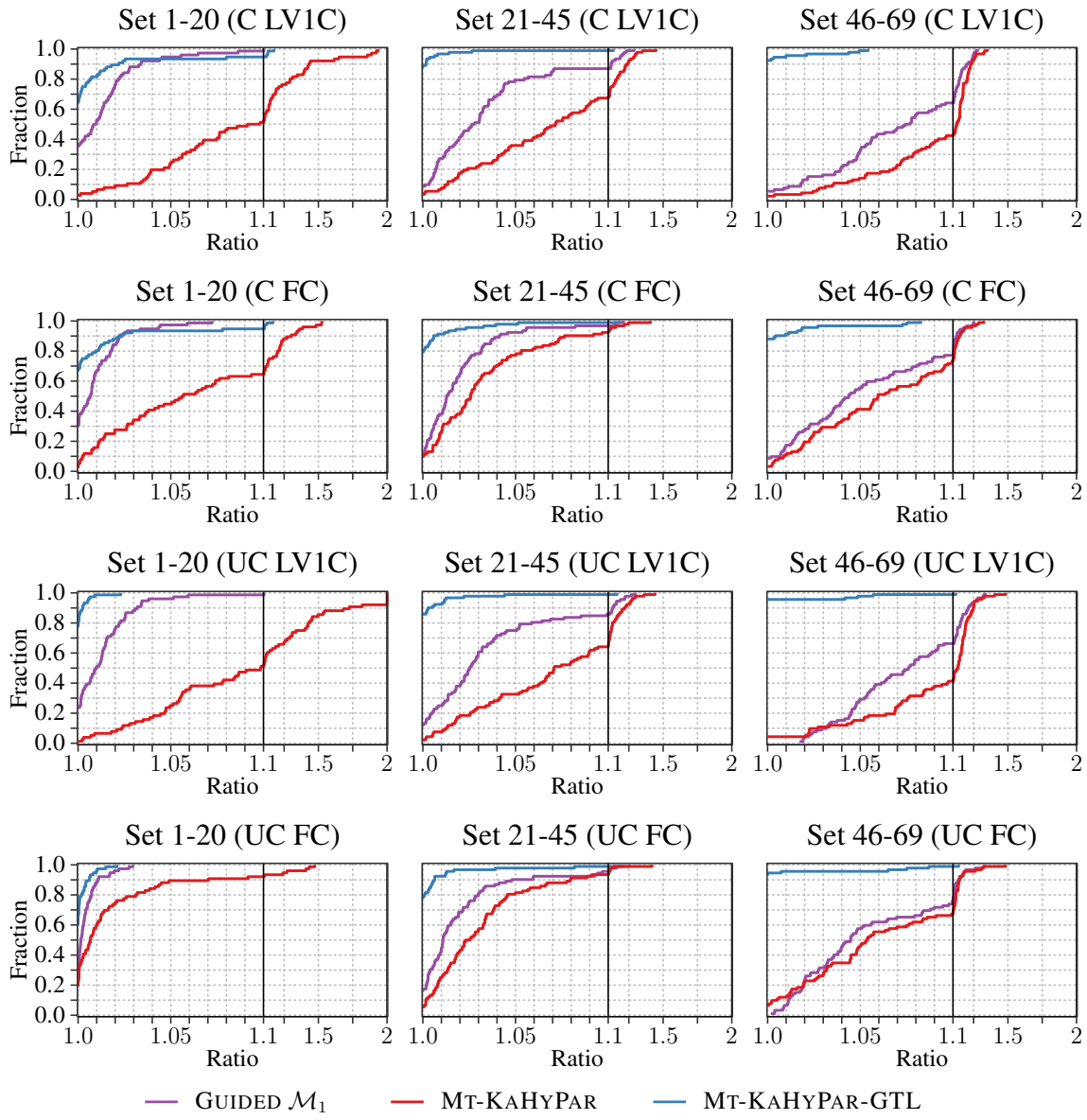

Figure 5.3: Partition quality of GUIDED $\mathcal{M}_1$ on three graph subsets under both constrained (C) and unconstrained (UC) refinement, evaluated at two stages: the Level 1 Cut (LV1C) and the Final Cut (FC). The left, middle, and right columns correspond to graphs 1-20, 21-45, and 46-69, respectively. A general trend of degraded partition quality is observed for graphs with lower mean values.

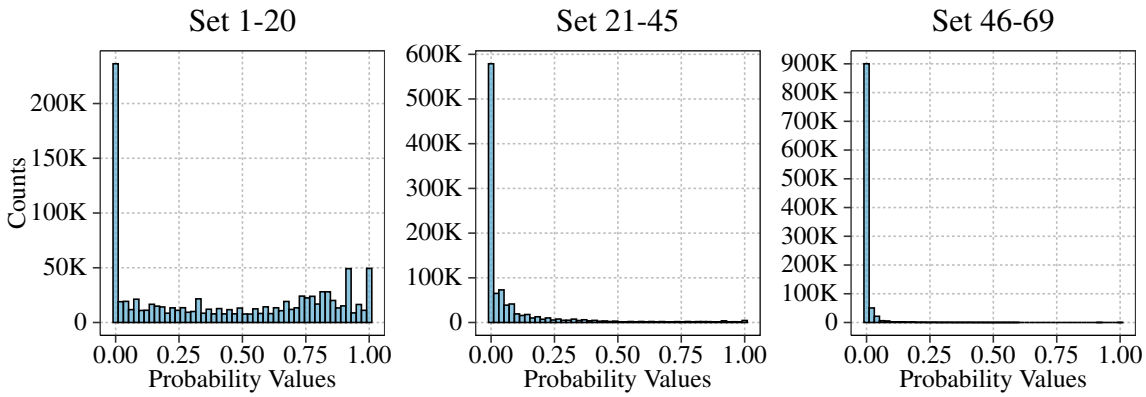

Figure 5.4: Distribution of the ground truth probabilities for the three graph subsets.

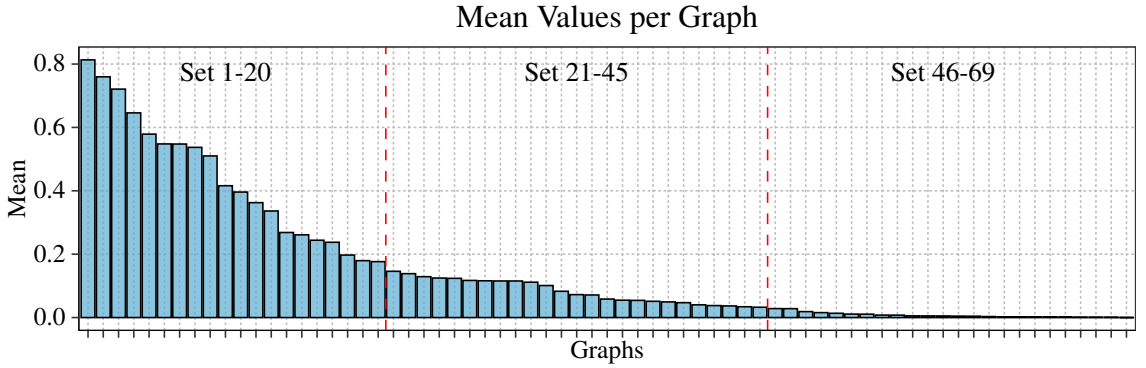

Figure 5.5: List of the sorted mean values μ of the 69 graphs.

$\mathcal{M}_1$ nearly matches the partition quality of MT-KAHYPAR-GTL in the final cut. Remarkably, in the constrained setting, $\mathcal{M}_1$ achieves a final partition with 0.03% fewer edge cuts compared to MT-KAHYPAR-GTL. These findings indicate that for graphs with less skewed distributions, the model effectively learns the mapping and achieves nearly all of the partition quality gains observed with the training labels guided algorithm. In contrast, the results also indicate that partition quality deteriorates as the mean probability decreases; that is, the model performs worse on graphs with lower mean probability values relative to the ground truth. Tables 5.3 and 5.4 further confirm the trend observed in Figure 5.3: partition quality consistently deteriorates in subsets characterized by lower mean values.

Refinement	Model	Reference Model	Level 1 Cut	Final Cut
UC	GUIDED $\mathcal{M}_1$	MT-KAHYPAR	-22.10%	-2.87%
C	GUIDED $\mathcal{M}_1$	MT-KAHYPAR	-15.28%	-9.12%
UC	GUIDED $\mathcal{M}_1$	MT-KAHYPAR-GTL	+1.23%	+0.26%
C	GUIDED $\mathcal{M}_1$	MT-KAHYPAR-GTL	+0.33%	-0.03%

Table 5.2: Difference of the edge cut amount of the GUIDED $\mathcal{M}_1$ algorithm in comparison to the unguided MT-KAHYPAR algorithm and the training labels guided MT-KAHYPAR-GTL algorithm on the first 20 graphs.

Refinement	Model	Reference Model	Level 1 Cut	Final Cut
UC	GUIDED $\mathcal{M}_1$	MT-KAHYPAR	-4.55%	-1.25%
C	GUIDED $\mathcal{M}_1$	MT-KAHYPAR	-5.01%	-1.73%
UC	GUIDED $\mathcal{M}_1$	MT-KAHYPAR-GTL	+3.93%	+1.74%
C	GUIDED $\mathcal{M}_1$	MT-KAHYPAR-GTL	+3.86%	+1.64%

Table 5.3: Difference of the edge cut amount of the GUIDED $\mathcal{M}_1$ algorithm in comparison to the unguided MT-KAHYPAR algorithm and the training labels guided MT-KAHYPAR-GTL algorithm on graph set 21-45.

Conversely, these findings indicate that the model has difficulty learning the mapping for graphs with highly skewed distributions. Consequently, the mixed results call for further investigation into the underlying causes and potential remedies, particularly with regard to unbalanced target distributions.

5.2.3 Unbalanced Distribution

To address the challenge of unbalanced target distributions, we employ two distinct strategies. First, we utilize a weighted loss function whose weights are determined by a density estimate of the underlying distribution. Second, we recompute the training labels using a different partitioner configuration, thereby mitigating imbalance in the data.

Refinement	Model	Reference Model	Level 1 Cut	Final Cut
UC	GUIDED $\mathcal{M}_1$	MT-KAHYPAR	-3.20%	-0.91%
C	GUIDED $\mathcal{M}_1$	MT-KAHYPAR	-3.60%	-1.36%
UC	GUIDED $\mathcal{M}_1$	MT-KAHYPAR-GTL	+8.65%	+5.75%
C	GUIDED $\mathcal{M}_1$	MT-KAHYPAR-GTL	+8.12%	+5.23%

Table 5.4: Difference of the edge cut amount of the GUIDED $\mathcal{M}_1$ algorithm in comparison to the unguided MT-KAHYPAR algorithm and the training labels guided MT-KAHYPAR-GTL algorithm on graph set 46-69.

Density-Based Weighting

The first strategy incorporates a weighted loss function during training. This approach is analogous to methods used in binary classification tasks for imbalanced classes, where miss classification errors for underrepresented classes are penalized more heavily than those for overrepresented ones. In regression tasks, however, the continuous nature of the target variable complicates direct adaptation of these techniques, as there are no discrete class labels to balance.

To overcome this, we adopt a density-based weighting scheme [33]. In this approach, the probability density function of the target variable is first estimated using kernel density estimation (KDE). This estimated density is then normalized to yield values between 0 and 1, where a value of 1 corresponds to the most frequent target values and 0 to the rarest. Each data point is assigned a weight that is inversely proportional to its normalized density, ensuring that data points from underrepresented regions of the target distribution contribute more significantly to the loss function. Consequently, the model is encouraged to minimize errors particularly in these rare, yet often critical, regions. The hyperparameter α allows us to control the scaling of the density influence. Larger values of α increase the weight of samples in less dense regions.

To evaluate the density-based weighting method, we employed the same hyperparameters as in model $\mathcal{M}_1$ and subsequently integrated the density weights modulated by the hyperparameter α. We experimented with various α values and compared the resulting partition quality to that of model $\mathcal{M}_1$. Specifically, the density weight model with $\alpha = 2$ is denoted as model $\mathcal{M}_2$ (see Figure 5.6), and the model with $\alpha = 50$ is denoted as model $\mathcal{M}_3$ (see Figure 5.7). Despite using very high α values, no significant partition quality improvement was achieved. Table 5.5 indicates that model $\mathcal{M}_3$ performs slightly worse overall, while model $\mathcal{M}_2$ shows inferior partition quality in the unconstrained scenario and only marginal gains with constrained refinement.

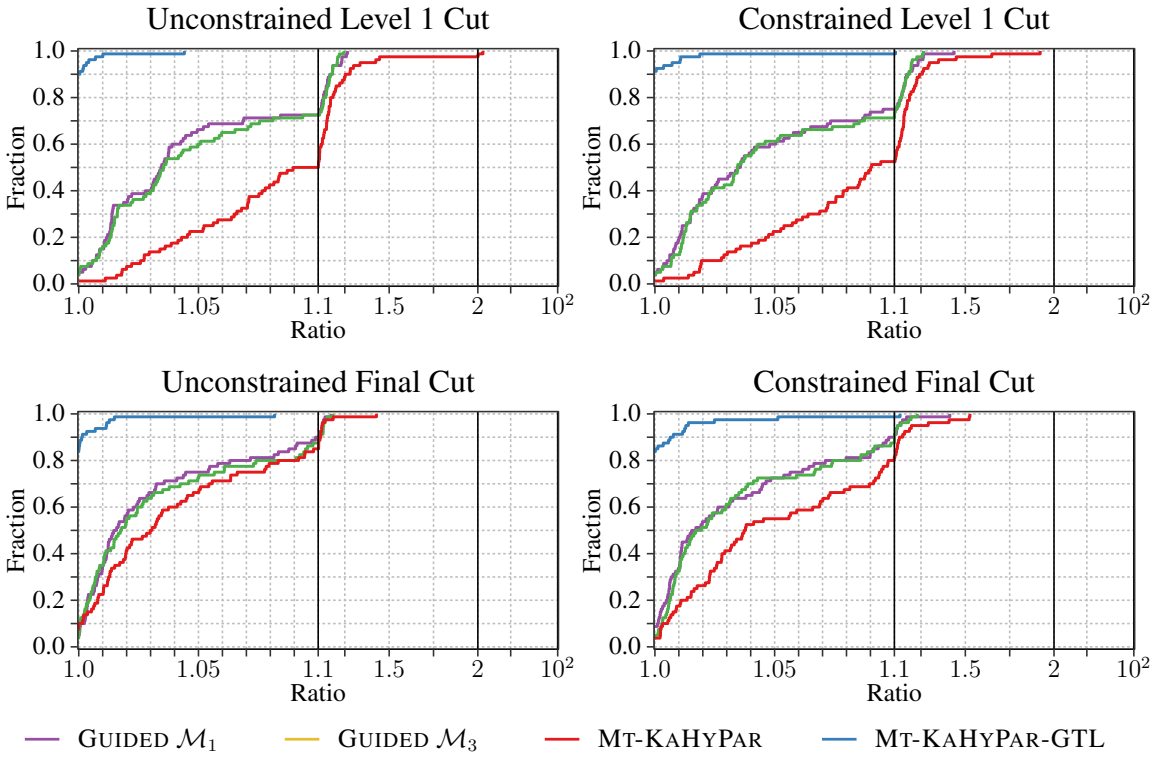

Figure 5.6: Partition quality of the density weight model (GUIDED $\mathcal{M}_2$) with $\alpha = 2$.

Refinement	Model	Reference Model	Level 1 Cut	Final Cut
UC	GUIDED $\mathcal{M}_2$	GUIDED $\mathcal{M}_1$	+0.22%	+0.29%
C	GUIDED $\mathcal{M}_2$	GUIDED $\mathcal{M}_1$	-0.14%	-0.02%
UC	GUIDED $\mathcal{M}_3$	GUIDED $\mathcal{M}_1$	+0.38%	+0.32%
C	GUIDED $\mathcal{M}_3$	GUIDED $\mathcal{M}_1$	+0.20%	+0.23%

Table 5.5: Difference of the edge cut amount of the two denseweight models $\mathcal{M}_2$ and model $\mathcal{M}_3$ in comparison to the first model $\mathcal{M}_1$.

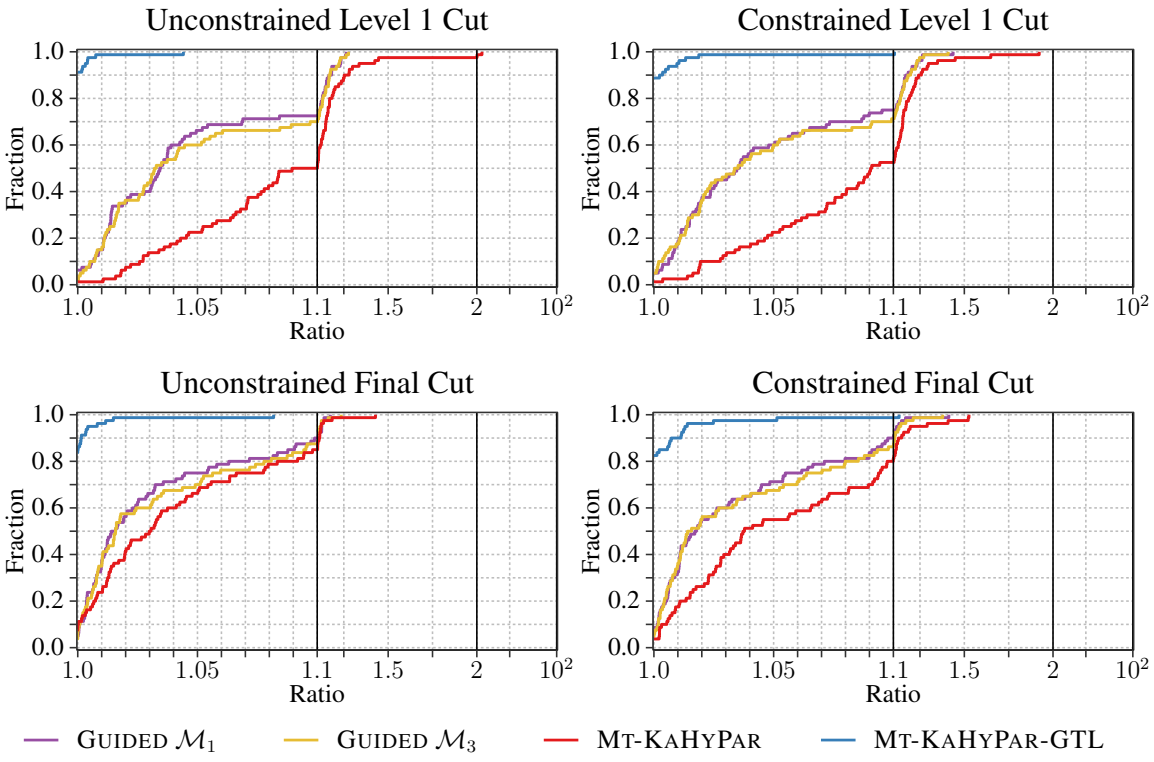

Figure 5.7: Partition quality of the density weight model (GUIDED $\mathcal{M}_3$) with $\alpha = 50$.

Label Balancing

The second strategy involves recomputing the labels to generate more balanced probability distributions. As previously discussed, the labels were computed by generating multiple high-quality partitions with different k values. When higher k values are used, the number of edges that must be cut in the final partition increases, which in turn results in a higher proportion of edges having probabilities greater than zero or even close to one. This effect is expected to increase the mean probability values and reduce the skewness of the distribution. For rebalancing, a subset of 56 graphs was selected from the original set of 69. A comparison between the distributions of the original and rebalanced labels for these 56 graphs reveals only a slight overall shift, with fewer samples in the bucket near zero and a modest increase in the other buckets (see Figure 5.9). But a detailed examination of the mean values for individual graphs (see Figure 5.8) indicates that, in some cases, the increase is almost a factor of 10.

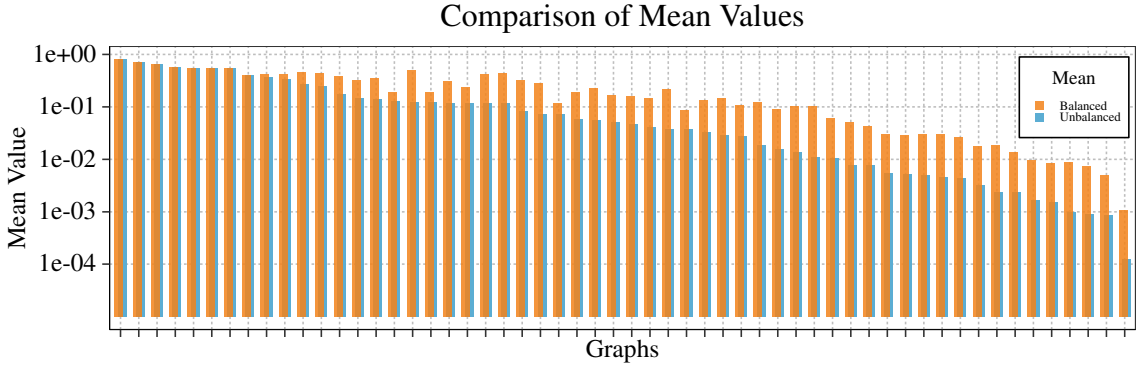

Figure 5.8: Mean values of the balanced 56 graphs in comparison the the unbalanced labels

Refinement	Model	Reference Model	Level 1 Cut	Final Cut
UC	GUIDED $\mathcal{M}_5$	GUIDED $\mathcal{M}_4$	-0.87%	+0.05%
C	GUIDED $\mathcal{M}_5$	GUIDED $\mathcal{M}_4$	-0.53%	+0.51%

Table 5.6: Difference of the edge cut amount of the balanced model $\mathcal{M}_5$ in comparison to the unbalanced model $\mathcal{M}_4$.

Subsequently, a model was trained using the newly computed labels and its partition quality was compared to that of an identical model trained on the original labels. The evaluation was performed on a benchmark set comprising 15 randomly sampled graphs from the 56-graph subset. The hyperparameters for both models were manually selected. The network architecture consisted of two hidden layers with 1024 neurons each, i.e., $\mathcal{A} = (208, 1024, 1024, 1)$. The models were trained for a maximum of $\tau = 200$ epochs on

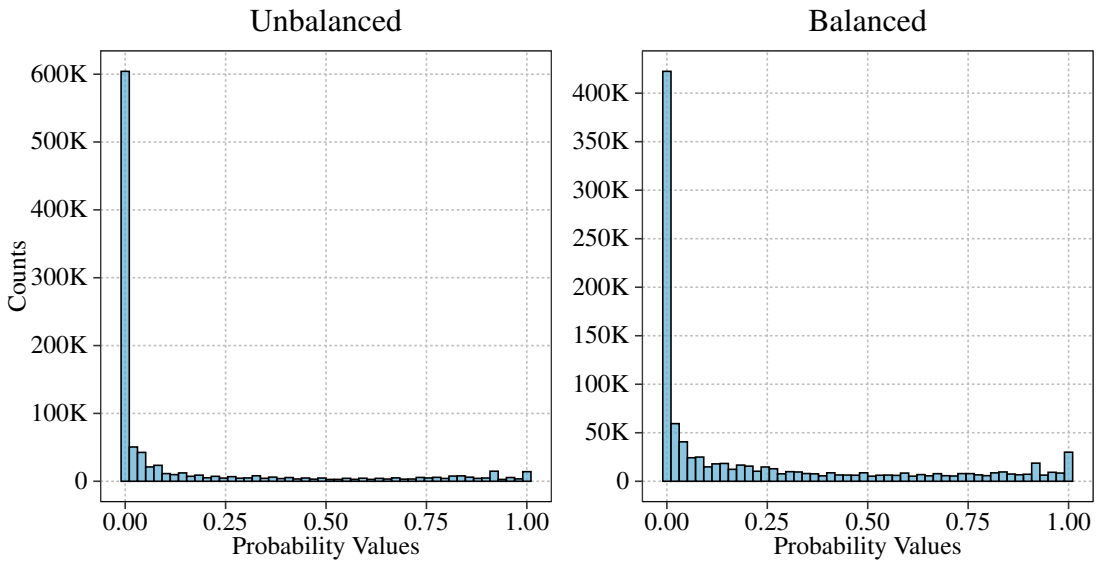

Figure 5.9: Distribution of the unbalanced and balanced labels of the set containing 56 graphs.

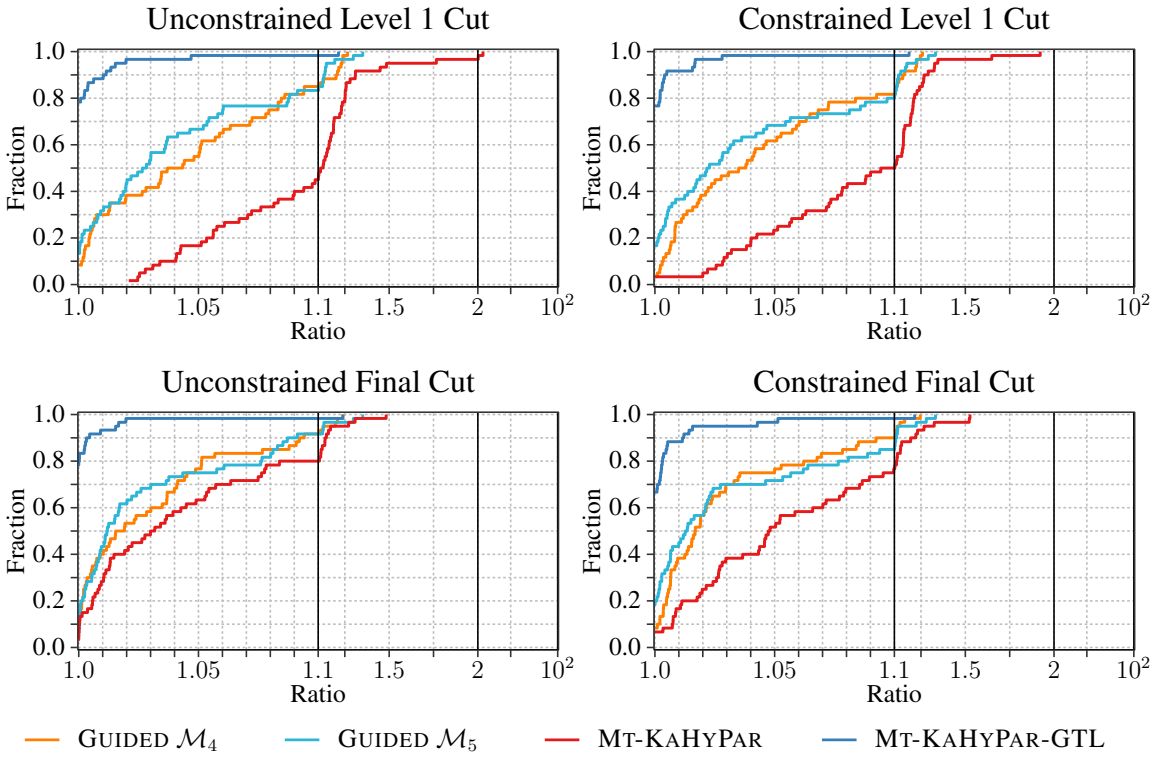

Figure 5.10: Partition quality of the two models trained on the graph set containing 57 graphs. While GUIDED $\mathcal{M}_4$ uses the old labels, GUIDED $\mathcal{M}_5$ uses the newly recomputed labels.

$a = 1\,000\,000$ data samples with a batch size of $\kappa = 128$. A starting learning rate of $\eta = 0.001$ was employed with batch normalization activated ($\beta = 1$). Regularization were as follows: dropout ($\delta = 0.25$), early stopping with a patience of $\rho = 20$, and weight decay ($\lambda = 0.000004$). We denote the model trained on the 56 graphs with the original labels as model $\mathcal{M}_4$ and the model trained with the rebalanced labels as model $\mathcal{M}_5$. The performance plots in Figure 5.10 show that the rebalanced labels lead to a small improvement in quality at the level 1 cuts, while the final cut exhibits either a decline in quality or almost no change. This trend is further evidenced by the results presented in Table 5.6.

Graphset	Refinement	Model	Reference Model	Level 1 Cut	Final Cut
Set 1-20	UC	GUIDED $\mathcal{M}_5$	GUIDED $\mathcal{M}_4$	-0.08%	-0.05%
	C	GUIDED $\mathcal{M}_5$	GUIDED $\mathcal{M}_4$	+0.05%	+0.14%
Set 21-45	UC	GUIDED $\mathcal{M}_5$	GUIDED $\mathcal{M}_4$	-0.76%	+0.11%
	C	GUIDED $\mathcal{M}_5$	GUIDED $\mathcal{M}_4$	-1.24%	-0.32%
Set 46-69	UC	GUIDED $\mathcal{M}_5$	GUIDED $\mathcal{M}_4$	-0.99%	+0.09%
	C	GUIDED $\mathcal{M}_5$	GUIDED $\mathcal{M}_4$	-0.60%	+0.38%

Table 5.7: Difference of the edge cut amount of the balanced model $\mathcal{M}_5$ in comparison to the unbalanced model $\mathcal{M}_4$, evaluated on the three graph subsets.

Based on the results, we conclude that the benchmark set sampled from all 56 graphs exhibits only minor changes. To further investigate the models behavior, we analyze the three subsets previously used for evaluating model $\mathcal{M}_1$, this time limited to the 56 graphs. We evaluated model $\mathcal{M}_5$, which was trained on these 56 graphs, alongside its reference model $\mathcal{M}_4$ across the three subsets. Unfortunately, the differences remained minimal. As shown in Table 5.7 and Figure 5.11, most performance improvements were observed only at the level 1 cut. On this level, the constrained refinement on set 21-45 reduced the number of cut edges by 1.24% compared to the reference model, while the unconstrained refinement on set 46-69 achieved a 0.99% reduction. Moreover, in the final cut, the configuration that had produced the most significant change at the level 1 cut resulted in 0.32% fewer cut edges, whereas the other configurations in the two lower mean subsets experienced a slight increase.

5.2.4 Dimensionality Reduction

To provide a more comprehensive evaluation, we also investigate different aspects of model performance. In addition to addressing distribution skewness, the subsequent experiments examine dimensionality reduction via Principal Component Analysis and assess feature cost. This integrated approach ensures that the study not only mitigates issues related to unbalanced distributions but also explores the benefits of reducing feature dimensionality on overall performance.

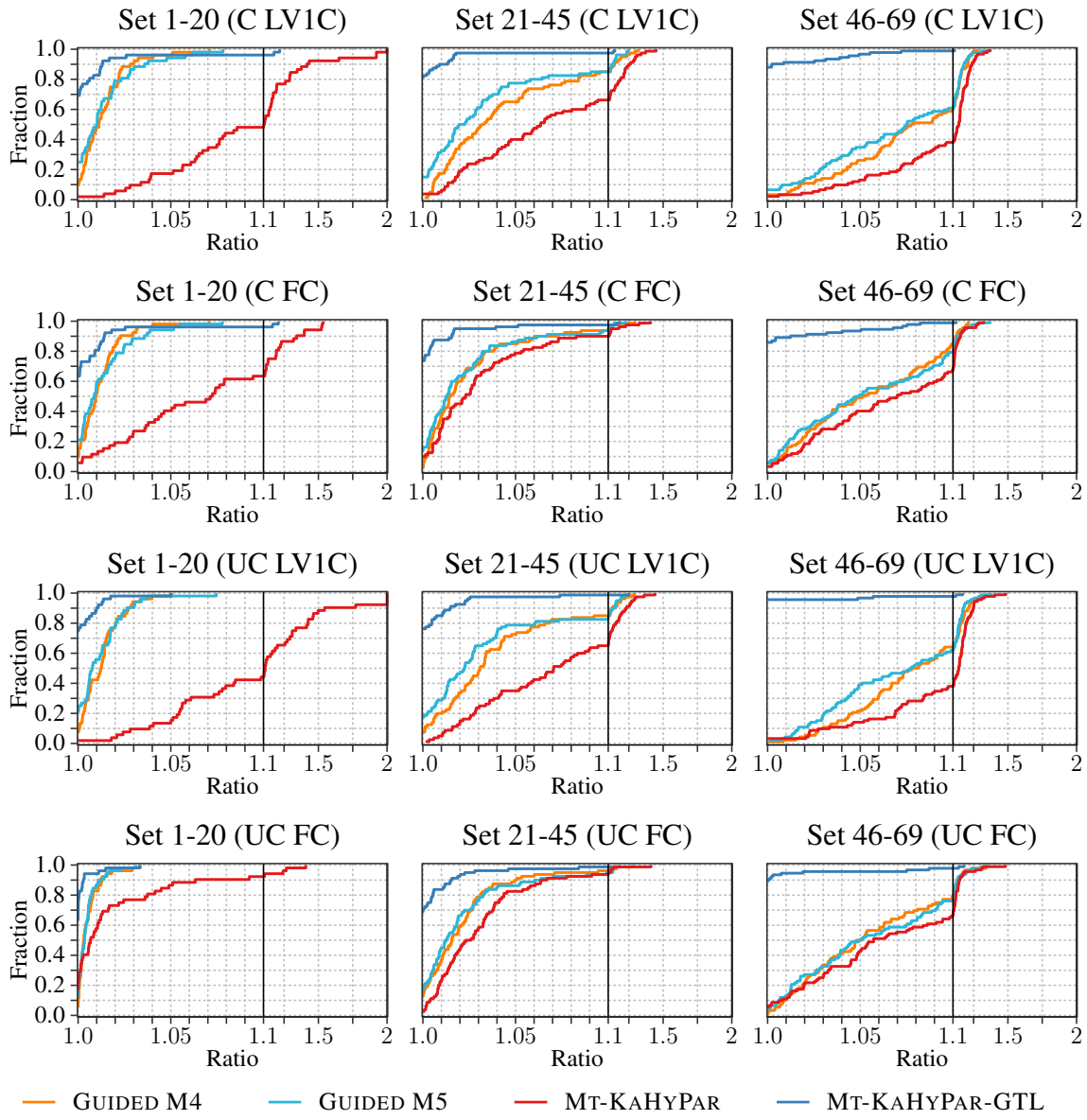

Figure 5.11: Partition quality of the two models trained on the graph set containing 57 graphs, evaluated on the three different subsets.

Principal Component Analysis

As PCA is a standard approach in deep neural network training, we applied it to the 208 features in order to reduce dimensionality and capture the most significant sources of variability. To determine the optimal number of principal components, we generated a scree plot[7] based on the features and selected the point at which the explained variance ratio ceased to decline significantly (see Figure 5.12). Subsequently, we trained a model using the same hyperparameters as model $\mathcal{M}_1$, reducing the feature space to 14 principal components. However, the results indicate that the partition quality slightly deteriorates with PCA-based dimensionality reduction (see Figure 5.13 and Table 5.8) with unconstrained refinement and does not change with constrained refinement.

Refinement	Model	Reference Model	Level 1 Cut	Final Cut
UC	GUIDED $\mathcal{M}_6$	GUIDED $\mathcal{M}_1$	+0.47%	+0.26%
C	GUIDED $\mathcal{M}_6$	GUIDED $\mathcal{M}_1$	+0.03%	+0.07%

Table 5.8: Difference of the edge cut amount of the model $\mathcal{M}_5$ that uses PCA in comparison to the first model $\mathcal{M}_4$.

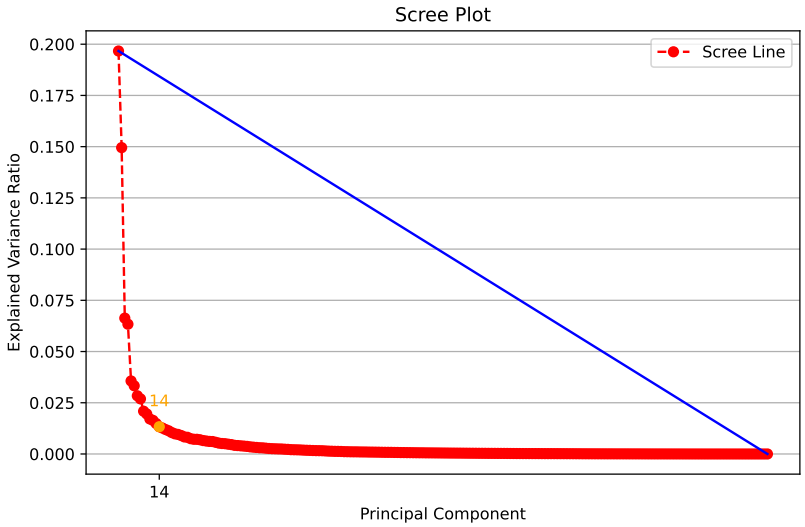

Figure 5.12: Scree plot for determining the optimal amount of principal components.

[7] https://en.wikipedia.org/wiki/Scree_plot

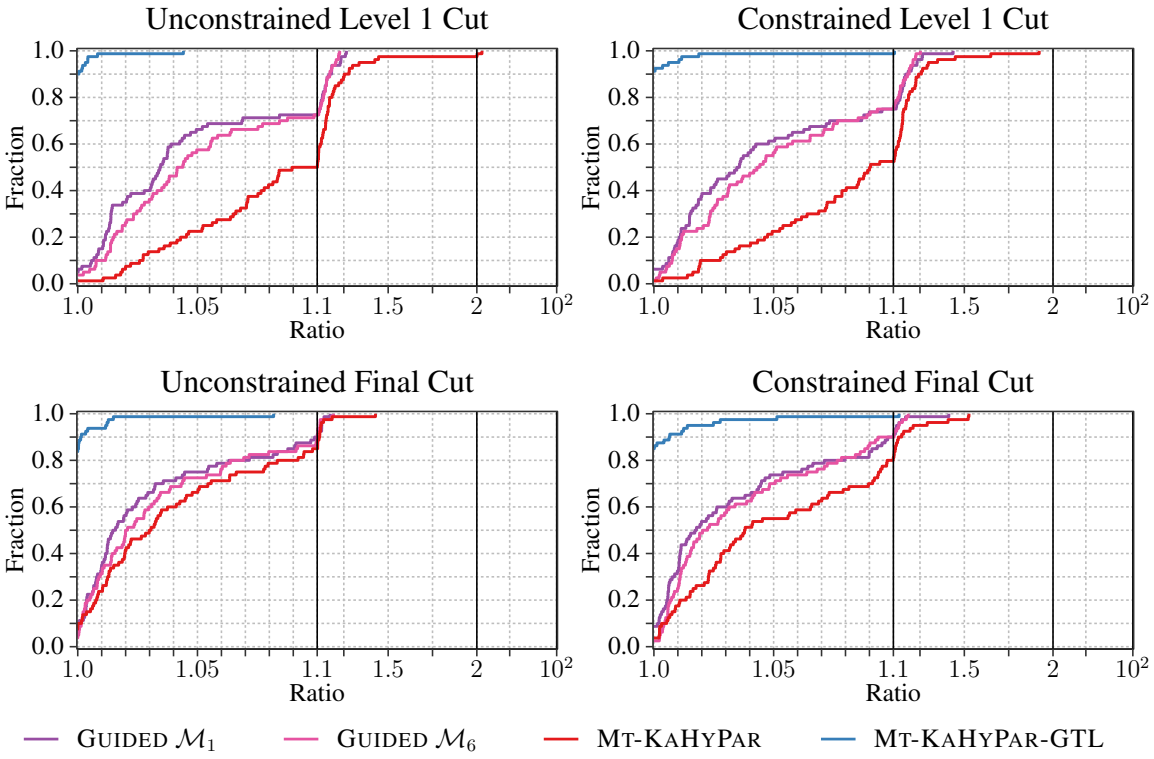

Figure 5.13: Partition quality of GUIDED $\mathcal{M}_6$ which used PCA to reduce its dimensionality in comparison to GUIDED $\mathcal{M}_1$.

Feature Reduction

As previously noted, feature computation is both timeconsuming and computationally expensive. Our analysis demonstrates that the 208 features employed in earlier sections significantly improve partition quality, particularly for graphs with higher mean values. This finding raises the question of whether reducing the number of features can lower computational costs while maintaining comparable partition quality gains.

To address this issue, we categorized the features according to their computational cost: features that are very easy to compute were assigned a cost of 0, those requiring moderate computation a cost of 1, and the most computationally intensive features a cost of 2 or higher. Subsequently, we trained a model using only the subset of features with low computational cost. Specifically, model $\mathcal{M}_7$ was trained on this feature subset, which contains 57 features, using the 20 graphs from the first of three subsets defined by the mean values μ of the graphs. The model's partition quality was evaluated on the same set of graphs, with the graph containing the most edges omitted from the evaluation. The hyperparameters for model $\mathcal{M}_7$ were manually selected and identical to those used for model $\mathcal{M}_4$ and model $\mathcal{M}_5$. The reference model $\mathcal{M}_8$ was also trained on the same data with the same hyperparameters but with all 208 features enabled.

Refinement	Model	Reference Model	Level 1 Cut	Final Cut
UC	GUIDED $\mathcal{M}_7$	GUIDED $\mathcal{M}_8$	+0.32%	+0.18%
C	GUIDED $\mathcal{M}_7$	GUIDED $\mathcal{M}_8$	+0.25%	-0.03%
UC	GUIDED $\mathcal{M}_7$	MT-KAHYPAR	-22.00%	-2.72%
C	GUIDED $\mathcal{M}_7$	MT-KAHYPAR	-15.37%	-9.38%

Table 5.9: Difference of the edge cut amount of the model $\mathcal{M}_7$ that was trained with the inexpensive features on graph set 1-20 in comparison to a reference model $\mathcal{M}_8$ that was trained with all features enabled and to the default configuration without guiding (MT-KAHYPAR).

The results demonstrate that the subset of computationally inexpensive features achieves performance levels nearly equivalent to the full feature set, with only a slight reduction in partition quality (see Figure 5.14). Moreover, Table 5.9 illustrates that even after feature reduction, the final partition exhibits 2.72% fewer edge cuts compared to the default unguided configuration. This observation suggests that there exist graphs for which partition quality is significantly enhanced even when low-cost features are used. Consequently, these findings indicate the potential for integrating such a model directly into the partitioner.

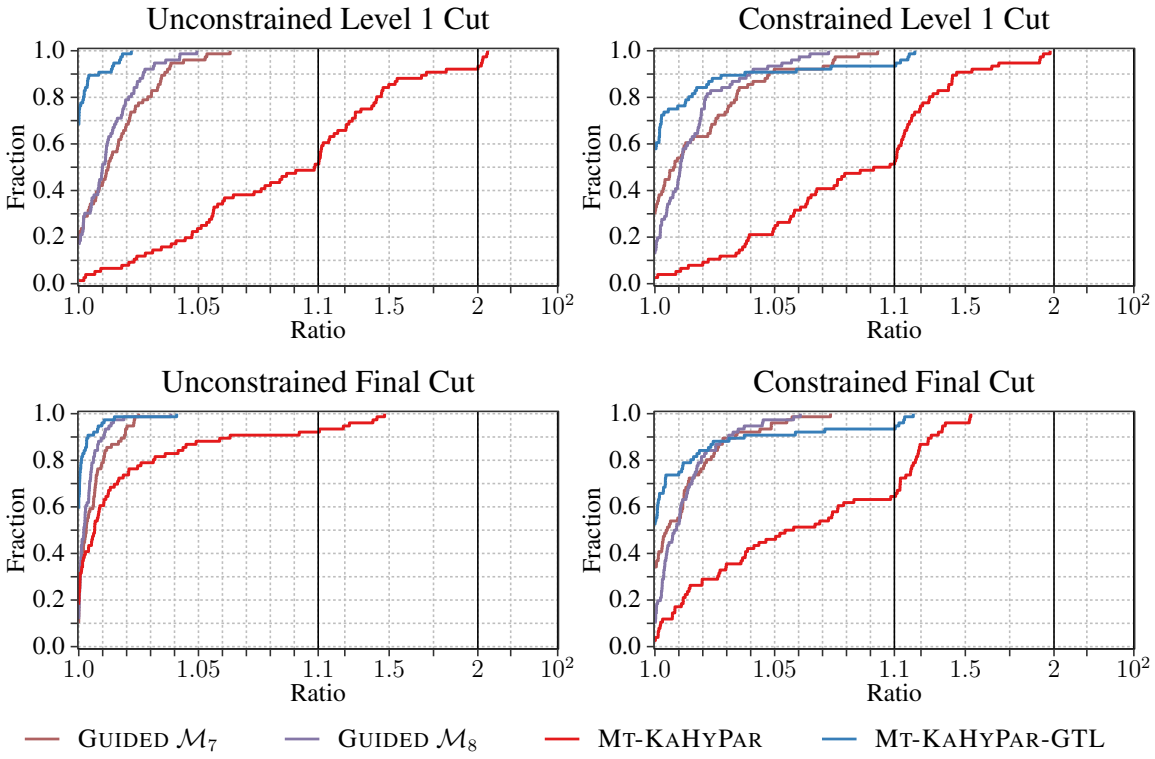

Figure 5.14: Partition quality of the model which was only trained on inexpensive features (GUIDED $\mathcal{M}_7$) in comparison to the model that was train with all features GUIDED $\mathcal{M}_8$).

6 Conclusion

In this thesis, we introduce a novel approach to improve the coarsening phase of the Multilevel Graph Partitioning (MGP) paradigm by leveraging machine learning. Specifically, we propose a deep neural network model to estimate the probability that a given edge would be cut in a high-quality final partition. These probabilities were then incorporated into a guided coarsening procedure, influencing the formation of clusters so that edges with high cut likelihood remain uncontracted. We evaluate the potential of this technique to improve partition quality by integrating it into an existing partitioner (Mt-KaHyPar).

Our experiments showed that guided coarsening can achieve improved partition quality for certain graph classes. In particular, graphs with relatively balanced cut-probability distributions benefited most from the additional information, yielding consistently smaller cut sizes compared to the baseline partitioner. In contrast, highly skewed probability distributions, where most edges have near-zero cut likelihood, posed challenges in training the model to identify the few crucial edges that should be left uncontracted. Nevertheless, even with skewed distributions, the model delivers a noticeable improvement in partition quality compared to the unguided baseline.

We also demonstrated that a reduced feature set, focusing on low-cost structural properties, can maintain similar partition quality gains in certain cases. This indicates a promising avenue for reducing feature-computation expenses, making direct integration of the model more feasible. Nonetheless, additional research is required to further identify improvements and limitations of the guided coarsening approach.

In summary, this machine-learning-guided coarsening approach paves the way for enhanced partition quality in multilevel methods particularly for graphs with moderately skewed probability distributions. With refined feature selection and improved handling of distributional imbalances, the technique could become a powerful component in large-scale graph-partitioning frameworks.

6.1 Future Work

Several directions emerge for future investigation:

Incorporating Graph Neural Networks. Recent work on integrating machine learning into traditional graph algorithms often leverages Graph Neural Networks (GNNs). Incorporating a GNN into our feature-based pipeline could potentially enhance accuracy,

especially when dealing with complex graph structures. Importantly, if the GNN component were to negatively impact performance, the model is designed to effectively deactivate it, reverting to the baseline solution. Thus, the integration of a GNN either improves performance or, at worst, maintains the current level without any detrimental effects.

Generalization to Unseen Data. Since our present experiments use the same graph instances for both training and final evaluation, a key next step is to eliminate this data overlap and analyze whether the model generalizes effectively to genuinely unseen graphs.

Addressing Label Imbalance. Although our study investigated multiple methods for addressing skewed label distributions, future research could explore additional techniques. One particular approach is SMOGN [5], a sampling method that generates synthetic minority samples to improve a model's predictive performance, especially in underrepresented regions of the dataset. By artificially adjusting the label distribution, SMOGN serves as a strategy to mitigate imbalance and enhance overall model accuracy.

Extending to Hypergraphs and Weighted Graphs. Our current approach applies only to unweighted plain graphs, limiting its scope. An extension to hypergraphs and to graphs with weights other than 1, would enable solutions to a broader range of partitioning problems, thereby expanding the practical usefulness of this method.

Lightweight Feature Computation. Finally, focusing on a carefully chosen subset of features, along with an efficient method for computing them, can substantially reduce overhead, making it feasible to integrate the model directly into the partitioner. This careful feature selection is crucial for balancing runtime cost with partition quality. Ensuring minimal performance impact while preserving strong partition outcomes remains a key challenge.

Bibliography

[1] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-quality shared-memory graph partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):27102722, November 2020.

[2] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA04, page 120124. ACM, June 2004.

[3] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[4] 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, October 2008.

[5] Paula Branco, Luís Torgo, and Rita P Ribeiro. Smogn: a pre-processing approach for imbalanced regression. In *First international workshop on learning with imbalanced domains: Theory and applications*, pages 36–50. PMLR, 2017.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201213, January 2002.

[8] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, DAC88. ACM, June 1988.

[9] Lars Gottesbüren and Michael Hamann. *Deterministic Parallel Hypergraph Partitioning*, page 301316. Springer International Publishing, 2022.

[10] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *ACM Transactions on Algorithms*, 20(1):154, January 2024.

[11] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. *Scalable Shared-Memory Hypergraph Partitioning*, page 1630. Society for Industrial and Applied Mathematics, January 2021.

[12] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep Multilevel Graph Partitioning. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2016.

[14] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 95*, Supercomputing 95, pages 28–es. ACM Press, 1995.

[15] Tobias Heuer. *A Direct k-Way Hypergraph Partitioning Algorithm for Optimizing the Steiner Tree Metric*, page 1531. Society for Industrial and Applied Mathematics, January 2024.

[16] Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[17] Sergey Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[18] Yu Jin, Andreas Loukas, and Joseph JaJa. Graph coarsening with preserved spectral properties. In *International Conference on Artificial Intelligence and Statistics*, pages 4452–4462. PMLR, 2020.

[19] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13:455–492, 1998.

[20] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

[21] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Application in vlsi domain. In *Proceedings of the 34th Design Automation Conference*, page 526529. IEEE, 1997.

[22] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, DAC-99, page 343348. IEEE, 1999.

[23] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359392, January 1998.

[24] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 225–236. IEEE, 2013.

[26] Nikolai Maas, Lars Gottesbüren, and Daniel Seemaier. *Parallel Unconstrained Local Search for Partitioning Irregular Graphs*, page 3245. Society for Industrial and Applied Mathematics, January 2024.

[27] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):26252638, September 2017.

[28] Siddheshwar Vilas Patil and Dinesh B. Kulkarni. K-way balanced graph partitioning for parallel computing. *Scalable Computing: Practice and Experience*, 22(4):413424, December 2021.

[29] Clara Pizzuti and Simona E. Rombo. Algorithms and tools for proteinprotein interaction networks clustering, with a special focus on population-based stochastic methods. *Bioinformatics*, 30(10):13431352, January 2014.

[30] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, page 779788. IEEE, June 2016.

[31] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533536, October 1986.

[32] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner,

Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484489, January 2016.

[33] Michael Steininger, Konstantin Kobs, Padraig Davidson, Anna Krause, and Andreas Hotho. Density-based weighting for imbalanced regression. *Machine Learning*, 110(8):21872211, July 2021.

A Appendix

A.1 LLM Support for Writing

The work of this thesis was supported by generative large language models. Specifically, the models by OpenAi (GPT 4o, GPT o1, GPT o3-mini, GPT o3-mini-high) and the models by Google (Gemini 2.0 Flash, 2.0 Flash Thinking, 2.0 Flash Thinking Experimental with apps, 1.5 Pro) were used by giving the models sentences and asking for improvement regarding academic writing and reading flow.

A.2 Software

The code can be found on Gitlab:
`https://gitlab.kit.edu/ucfoh/bachelor-thesis`

A.3 Features

A.3.1 Distribution Statistics

To effectively capture the characteristics of feature value sequences across all feature categories, we compute a standard set of descriptive statistics for each relevant sequence. These statistics provide key insights into the central tendency, dispersion, range, and shape of the distributions, offering a way to provide the model with useful information. For each sequence, we calculate these statistics, as detailed below.

Measures of **central tendency** include the **average** (mean), representing the typical value, and the **median**, a robust measure of the central value which is less sensitive to outliers.

Dispersion is quantified using the **standard deviation**, which measures the average spread of values around the mean. The **minimum** and **maximum** values define the range of the distribution. **Quartiles**, specifically the **first** and **third**, divide the sorted data into fourths and provide a robust measure of dispersion for the lower 25% and the upper 25% of the data.

The **shape** of the distribution is characterized by **skewness** and **entropy**. **Skewness** measures the asymmetry of the distribution; a value of zero indicates symmetry, while positive

and negative values indicate right- and left-skewed distributions, respectively. **Entropy** quantifies the randomness or uniformity of the distribution. Higher entropy indicates a more uniform distribution, while lower entropy suggests values are concentrated in a narrower range.

By computing this comprehensive suite of statistics for a value sequence, we obtain a nuanced characterization of the underlying distributions. This statistical approach allows our machine learning model to capture essential distributional properties. These distribution statistics are consistently applied in the subsequent sections of global, edge-specific, and vertex-specific features.

A.3.2 Global Graph Features

To effectively characterize the graphs for the training of the model, we employ a set of 28 global graph features. These features, computed once for each graph and thus invariant across its edges, aim to capture the essential macroscopic properties of the network structure. They provide a high-level, holistic description of the graph, encompassing its size, density, the statistical distributions of node degrees and edge localities, degree irregularity, and community organization.

Fundamental Graph Descriptors The most basic descriptors of a graph are its size metrics: the number of nodes and the number of edges. The feature denoted as **n** directly represents the total number of vertices in the graph. This provides a fundamental measure of the scale of the network. Complementing this, the feature **m** quantifies the total count of edges within the graph. They establish the foundational dimensions of the graph.

Degree and Locality Both the degree and the locality of the graph are value sequences. The **degree sequence**, which lists the degree of every node in the graph, is a primary descriptor of node connectivity. From this sequence, we derive the set of statistical features as described in A.3.1. These include the **average degree**, **standard deviation of degree**, **skewness of degree**, **entropy of degree**, **minimum degree**, **maximum degree**, **median degree**, and the **first and third quartiles of degree**.

The **edge locality sequence** gets its values from each edge of the graph. For this we use the locality computation based on triangles

$$\frac{d(u,v)}{\min\{d(u),d(v)\}-1}$$

where $d(u,v)$ is the amount of triangles that contain $\{u,v\} \in E$ as an Edge, and $d(u)$ is the degree of the vertex u. Analogous to the degree sequence, we compute a the set of statistical features over the set of edge locality values. These include the **average locality**, **standard deviation of locality** , **skewness of locality**, **entropy of locality** , **minimum locality** , **maximum locality**, **median locality** , and **first and third quartile of locality**.

Irregularity The **irregularity** feature, also known as degree heterogeneity, normalizes degree variability. It's calculated as the ratio of the degree sequence's standard deviation to its average:

$$\textbf{irregularity} = \frac{\textbf{standard deviation degree}}{\textbf{average degree}}$$

This ratio allows comparison of degree variance across graphs with different scales and densities. Higher irregularity indicates a more uneven degree distribution.

Community Structure To quantify community structure, we use features from a multilevel community detection algorithm, assessing both community count and quality.

The **number of communities** is represented by 3 features, indicating counts at hierarchy levels 0, 1, and 2. These levels reflect different partitioning resolutions.

Modularity, evaluating community strength, is also represented by 3 features, one for each level (0, 1, 2). Higher modularity scores indicate a stronger community structure.

Expected Median Degree of Neighborhood The **expected median degree** feature measures typical local connectivity by representing the expected median degree of a node's neighborhood. This global feature provides insight into characteristic local neighborhood density, offering a nuanced view of graph connectivity beyond average and median degrees.

A.3.3 Vertex Features

A list of the vertex features can be found in Table A.1.

A.3.4 Edge Features

A list of the edge features can be found in Table A.2.

A.3.5 Feature Cost

A list of features with feature cost 0 can be found in Table A.3.

A.4 Graph Instances

Figure A.1 shows the names and their mean values in log 10 scale. Table A.4 shows the irregular graphs with their n, m, irregularity and average degree values. Table A.5 shows the regular graphs with the same features.

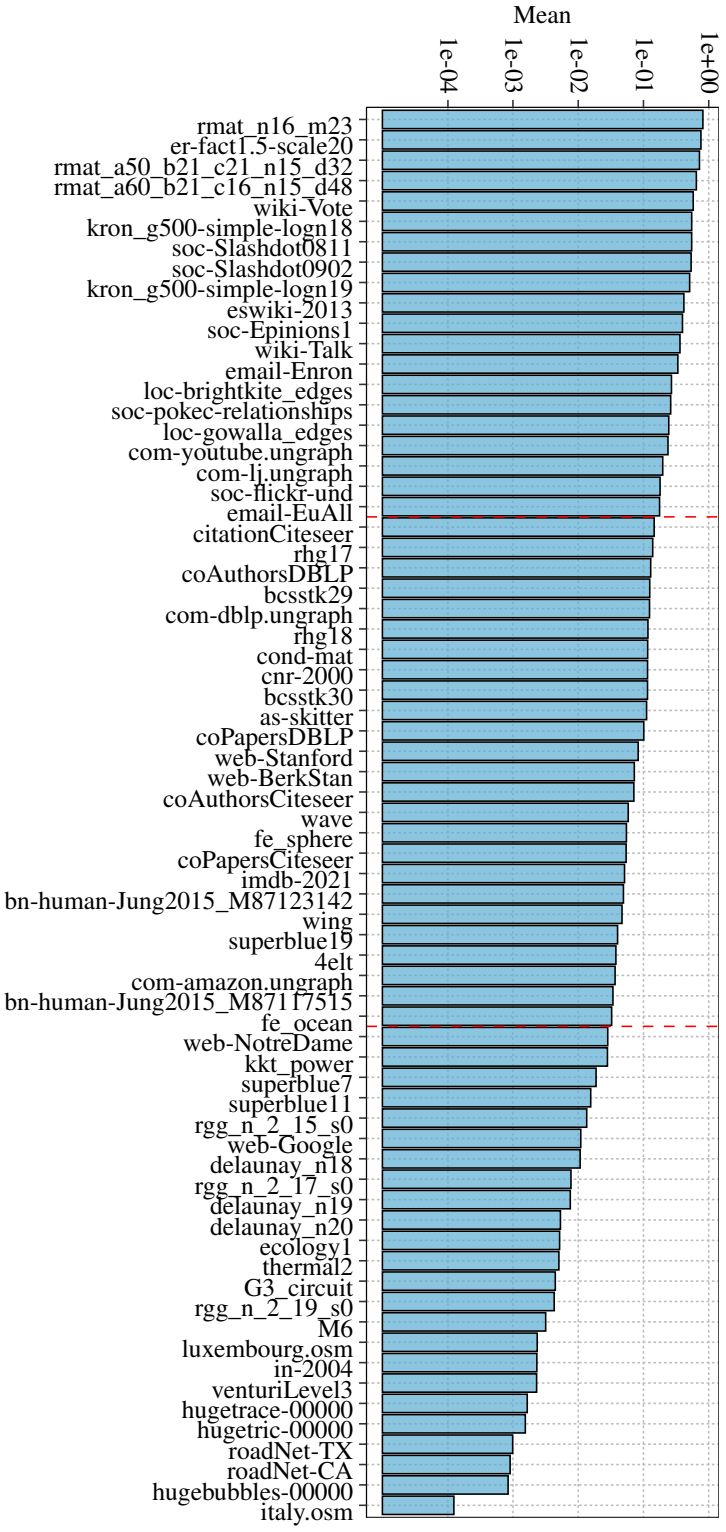

Figure A.1: A list of the 69 graphs with their respective mean values in log scale.

vertex features	
degree	n1_clustering_coefficient
degree_quantile	chi_squared_degree_deviation
avg_n1_degree	n1_max_clique
sd_n1_degree	n2_size
skew_n1_degree	avg_n2_degree
entropy_n1_degree	sd_n2_degree
min_n1_degree	skew_n2_degree
q1_n1_degree	entropy_n2_degree
med_n1_degree	min_n2_degree
q3_n1_degree	q1_n2_degree
max_n1_degree	med_n2_degree
avg_n1_locality	q3_n2_degree
sd_n1_locality	max_n2_degree
skew_n1_locality	avg_n2_locality
entropy_n1_locality	sd_n2_locality
min_n1_locality	skew_n2_locality
q1_n1_locality	entropy_n2_locality
med_n1_locality	min_n2_locality
q3_n1_locality	q1_n2_locality
max_n1_locality	med_n2_locality
n1_to_n1_edges	q3_n2_locality
n1_to_n2_edges	max_n2_locality
n1_d1_nodes	n2_to_n1n2_edges
n1_modularity	n2_out_edges
n1_max_modularity	n2_modularity
n1_max_modularity_size	n2_max_modularity
n1_min_contracted_degree	n2_max_modularity_size
n1_min_contracted_degree_size	

Table A.1: List of vertex features

edge features	
locality	intersect_n1_max_clique
jaccard_index	union_degree
cosine_similarity	union_degree_quantile
dice_similarity	union_avg_n1_degree
strawman_similarity	union_sd_n1_degree
intersect_degree	union_skew_n1_degree
intersect_degree_quantile	union_entropy_n1_degree
intersect_avg_n1_degree	union_min_n1_degree
intersect_sd_n1_degree	union_q1_n1_degree
intersect_skew_n1_degree	union_med_n1_degree
intersect_entropy_n1_degree	union_q3_n1_degree
intersect_min_n1_degree	union_max_n1_degree
intersect_q1_n1_degree	union_avg_n1_locality
intersect_med_n1_degree	union_sd_n1_locality
intersect_q3_n1_degree	union_skew_n1_locality
intersect_max_n1_degree	union_entropy_n1_locality
intersect_avg_n1_locality	union_min_n1_locality
intersect_sd_n1_locality	union_q1_n1_locality
intersect_skew_n1_locality	union_med_n1_locality
intersect_entropy_n1_locality	union_q3_n1_locality
intersect_min_n1_locality	union_max_n1_locality
intersect_q1_n1_locality	union_n1_to_n1_edges
intersect_med_n1_locality	union_n1_to_n2_edges
intersect_q3_n1_locality	union_n1_d1_nodes
intersect_max_n1_locality	union_n1_modularity
intersect_n1_to_n1_edges	union_n1_max_modularity
intersect_n1_to_n2_edges	union_n1_max_modularity_size
intersect_n1_d1_nodes	union_n1_min_contracted_degree
intersect_n1_modularity	union_n1_min_contracted_degree_size
intersect_n1_max_modularity	union_n1_clustering_coefficient
intersect_n1_max_modularity_size	union_chi_squared_degree_deviation
intersect_n1_min_contracted_degree	union_n1_max_clique
intersect_n1_min_contracted_degree_size	comm_0_equal
intersect_n1_clustering_coefficient	comm_1_equal
intersect_chi_squared_degree_deviation	comm_2_equal

Table A.2: List of edge features

features	cost	features	cost
n	0	min_n1_degree	0
m	0	q1_n1_degree	0
irregularity	0	med_n1_degree	0
exp_median_degree	0	q3_n1_degree	0
avg_degree	0	max_n1_degree	0
sd_degree	0	n1_to_n1_edges	0
skew_degree	0	n1_to_n2_edges	0
entropy_degree	0	n1_d1_nodes	0
min_degree	0	n1_modularity	0
q1_degree	0	n1_min_contracted_degree	0
med_degree	0	n1_min_contracted_degree_size	0
q3_degree	0	degree.1	0
max_degree	0	degree_quantile.1	0
n_communities_0	0	avg_n1_degree.1	0
n_communities_1	0	sd_n1_degree.1	0
n_communities_2	0	skew_n1_degree.1	0
modularity_0	0	entropy_n1_degree.1	0
modularity_1	0	min_n1_degree.1	0
modularity_2	0	q1_n1_degree.1	0
strawman_similarity	0	med_n1_degree.1	0
comm_0_equal	0	q3_n1_degree.1	0
comm_1_equal	0	max_n1_degree.1	0
comm_2_equal	0	n1_to_n1_edges.1	0
degree	0	n1_to_n2_edges.1	0
degree_quantile	0	n1_d1_nodes.1	0
avg_n1_degree	0	n1_modularity.1	0
sd_n1_degree	0	n1_min_contracted_degree.1	0
skew_n1_degree	0	n1_min_contracted_degree_size.1	0
entropy_n1_degree	0		

Table A.3: List of features with cost 0. Features marked with the suffix .1 originate from vertex 2.

class	graph	n	m	irregularity	avg_degree
Irregular Real-world	as-skitter	1696415	11095298	10.460000	13.080000
	citationCiteseer	268495	1156647	1.890000	8.616000
	coAuthorsCiteseer	227320	814134	1.485000	7.163000
	coAuthorsDBLP	299067	977676	1.501000	6.538000
	coPapersCiteseer	434102	16036720	1.371000	73.880000
	coPapersDBLP	540486	15245729	1.174000	56.410000
	com-amazon.ungraph	548552	925872	1.555000	3.376000
	com-dblp.ungraph	425957	1049866	1.847000	4.929000
	com-lj.ungraph	4036538	34681189	2.490000	17.180000
	com-youtube.ungraph	1157828	2987624	9.738000	5.161000
	email-Enron	36692	183831	3.603000	10.020000
	email-EuAll	265009	364481	13.930000	2.751000
	eswiki-2013	972933	21184931	10.660000	43.550000
	loc-brightkite_edges	58228	214078	2.768000	7.353000
	loc-gowalla_edges	196591	950327	5.542000	9.668000
	soc-Epinions1	75879	405740	4.024000	10.690000
	soc-Slashdot0811	77360	469180	3.335000	12.130000
	soc-Slashdot0902	82168	504230	3.346000	12.270000
	soc-pokec-relationships	1632803	22301964	1.569000	27.320000
	soc-flickr-und	1715255	15555041	7.146000	18.140000
	web-BerkStan	685230	6649470	14.690000	19.410000
	web-Google	875713	4322051	4.019000	9.871000
	web-NotreDame	325729	1090108	6.398000	6.693000
	web-Stanford	281903	1992636	11.790000	14.140000
	wiki-Talk	2394385	4659565	26.340000	3.892000
	wiki-Vote	7115	100762	2.033000	28.320000
	bn-human-Jung2015_M87117515	891589	48669606	1.211000	109.200000
	bn-human-Jung2015_M87123142	846535	53825327	1.663000	127.200000
	cnr-2000	325557	2738969	13.020000	16.830000
	imdb-2021	2996317	5369472	3.994000	3.584000
Irregular Artificial	kron_g500-simple-logn18	262144	10582686	5.619000	80.740000
	kron_g500-simple-logn19	524288	21780787	6.510000	83.090000
	rhg17	121189	484661	9.671000	7.998000
	rhg18	241527	987492	10.380000	8.177000
	rmat_n16_m23	65484	8388607	1.922000	256.200000
	rmat_a50_b21_c21_n15_d32	32768	973456	2.733000	59.420000
	rmat_a60_b21_c16_n15_d48	32768	1111980	4.222000	67.870000

Table A.4: Irregular Graphs

class	graph	n	m	irregularity	avg_degree
Regular Real-world	4elt	15606	45878	0.097530	5.880000
	G3_circuit	1585478	3037674	0.167500	3.832000
	M6	3501776	10501936	0.140800	5.998000
	cond-mat	16726	47594	1.128000	5.691000
	ecology1	1000000	1998000	0.015810	3.996000
	fe_ocean	143437	409593	0.117000	5.711000
	fe_sphere	16386	49152	0.006378	5.999000
	hugebubbles-00000	18318143	27470081	0.009301	2.999000
	hugetrace-00000	4588484	6879133	0.013190	2.998000
	hugetric-00000	5824554	8733523	0.011230	2.999000
	in-2004	1382908	13591473	7.456000	19.660000
	italy.osm	6686493	7013978	0.194600	2.098000
	kkt_power	2063494	6482320	1.186000	6.283000
	luxembourg.osm	114599	119666	0.196900	2.088000
	roadNet-CA	1971281	2766607	0.358100	2.807000
	roadNet-TX	1393383	1921660	0.376000	2.758000
	superblue11	1888238	3069269	2.733000	3.251000
	superblue19	1034167	1713796	5.905000	3.314000
	superblue7	2700635	4931418	3.289000	3.652000
	thermal2	1227087	3676134	0.132600	5.992000
	venturiLevel3	4026819	8054237	0.029560	4.000000
	wave	156317	1059331	0.245000	13.550000
	wing	62032	121544	0.070940	3.919000
	bcsstk29	13992	302748	0.361400	43.270000
	bcsstk30	28924	1007284	0.455500	69.650000
Regular Artificial	delaunay_n18	262144	786396	0.223500	6.000000
	delaunay_n19	524288	1572823	0.223100	6.000000
	delaunay_n20	1048576	3145686	0.222800	6.000000
	er-fact1.5-scale20	1048576	10904496	0.219300	20.800000
	rgg_n_2_15_s0	32768	160240	0.322600	9.780000
	rgg_n_2_17_s0	131072	728753	0.300700	11.120000
	rgg_n_2_19_s0	524288	3269766	0.284100	12.470000

Table A.5: Regular Graphs