Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Machine Learning for Graph Partitioning

Bachelor Thesis

Anton Schäfer

February 2, 2021

Supervisor: Prof. Dr. Torsten Hoefler
Advisors: Dr. Nikoli Dryden, Lukas Gianinazzi, Maciej Besta

Scalable Parallel Computing Lab, ETH Zürich

Acknowledgements

I would like to thank Prof. Dr. Torsten Hoefler and the Scalable Parallel Computing Lab for providing me with the opportunity and resources to write my thesis in this interesting area of research.

I am especially grateful for the support by my advisors Lukas Gianinazzi, Dr. Nikoli Dryden, and Maciej Besta. They guided me in approaching the topic, structuring my research and contributed many valuable ideas. It was a pleasure working with them.

Abstract

Balanced Graph Partitioning, i.e. the partitioning of a graph into k partitions of roughly equal size while minimizing the number of crossing edges, is a highly relevant combinatorial optimization problem due its applications in parallel processing and other fields. As the problem is NP-hard, it is usually solved approximately using specialized heuristic algorithms. These heuristics are hand-engineered to enable good solutions in short time for real-world inputs. For a variety of combinatorial optimization problems, machine learning methods have recently been applied succesfully to learn such heuristics.

In this work, we propose two approaches for using machine learning in balanced graph partitioning algorithms. We consider general featureless graphs and propose a new model architecture for generating node embeddings which proves useful for both approaches. The first approach relies entirely on an end-to-end machine learning model that assigns each node to a partition in a classification-style setup. The second approach is based on a meta algorithm that iteratively improves a given partitioning by swapping nodes' partition assignments. These nodes are selected according to a learned heuristic. To compute this heuristic, we use a neural network that is trained via Deep Q-Learning. We achieve an asymptotic speedup by splitting the Q-function in two parts.

Both approaches outperform the random baseline by a large margin. The simpler end-to-end approach performs consistently worse than the algorithmic baselines, while the more algorithmically oriented iterative improvement approach produces better partitionings than a greedy algorithm and is close in performance to more specialized solvers. We see potential for significant improvements upon these results in future work builiding on our methods.

Contents

Chapter 1

Introduction

Machine learning methods are often applied in fields in which humans have traditionally performed better than computers. They have recently achieved significant breakthroughs in computer vision and natural language processing, where knowledge about data distributions is highly important to solve problems successfully. In principle, however, machine learning can be useful in any setting where statistical patterns can be leveraged. This is often the case for combinatorial problems. Many practically employed algorithms for combinatorial problems make use of heuristics to speed up or guide the problem solving process. Even if these heuristics do not work well on all possible problem inputs and could potentially negatively impact performance, they are very useful in practice. Real-world data usually follows some distribution that can be exploited and, in practice, worst case inputs occur very rarely. Traditionally, such heuristics are hand-engineered, using human knowledge and (often implicit) assumptions about the data distribution. Machine learning models, however, can learn directly from the data and can therefore naturally be applied in this setting.

For various combinatorial problems, learned models have recently been shown to outperform existing hand-crafted heuristics: For games like Chess and Go, search methods guided by learned heuristics beat even the strongest algorithmic engines [52, 53]. Sorting algorithms with learned heuristics run faster than the best conventional sorting algorithms [38]. Even entire database systems where learned heuristics replace core components have been proposed [36, 37]. Especially NP-hard combinatorial optimization problems are often solved by specialized heuristic solvers, but lately, promising machine learning based methods have been developed for many such problems [39, 40, 35, 44].

In this work, we explore the use of such methods for balanced graph partitioning. Balanced graph partitioning is a hard combinatorial optimization problem of high practical relevance due to its applications in parallel processing and other fields. Given a graph, the goal is to partition its vertices into k partitions of roughly equal size while minimizing the edge cut, i.e. the number of edges between two partitions. Due to its computational hardness, the problem cannot be solved optimally even for moderately sized graphs, unless $P = NP$ [16, 24]. In practice, it is therefore solved using specialized heuristic algorithmic solvers that produce good, but not necessarily optimal solutions. Motivated by the success of machine learning and its recent applications to combinatorial optimization problems, we see potential for machine learning based graph partitioning methods. In this work, we propose algorithms for balanced graph partitioning that rely on learned heuristics.

1.1 Challenges

In the following, we discuss the major challenges associated with designing such methods.

Meta-algorithm. Algorithms for combinatorial optimization problems that rely on machine learning often employ a meta-algorithm that guides the problem solving process but relies on learned heuristics for making decisions. When designing such algorithms, it is essential to carefully select where machine learning models might be applied and which tasks or computations are better solved algorithmically. While algorithms are useful for cheap combinatorial computations that require exact results, machine learning models are particularly good at picking up on statistical patterns in data or computations. This makes them suitable for approximating expensive computations that can be used as heuristics. However, if the results of these computations do not follow any learnable statistical patterns, machine learning might not be beneficial. A suitable combination of meta-algorithm and machine learning models is thus essential for the success of the method. As an extreme approach, the meta-algorithm may be cut out completely, resulting in an end-to-end machine learning model that takes in a problem instance and outputs a solution.

Model architecture. Given a meta algorithm with a well defined task that is to be solved with machine learning methods, we need a suitable machine learning model. Such a model requires input features that contain enough information for deriving the desired outputs. When operating on abstract graphs without node features, it is not obvious how to engineer input features that are both meaningful and efficiently obtainable. Furthermore, the model architecture has to be expressive enough to map such inputs to the respective outputs. For successful training, the model should be parameterized in a way that allows for easily finding a good set of parameters.

Training algorithm. There are three main approaches to consider for training the model, each of which has advantages and disadvantages.

When taking a supervised approach, the model is trained on target labels. Such target labels may either be optimal ground truth labels or labels generated by a heuristic that the model should learn to imitate. However, when using non-optimal heuristic labels, the model can only ever be as good as the heuristic it is learning from. Hence, this method is only useful if the model forward pass can be computed more efficiently than the function it is supposed to approximate. This may be the case for particularly expensive heuristics or optimal ground truth labels which are extremely costly to compute in the NP-hard case. For the latter, it is especially important that the model is able to generalize from smaller problem instances to larger ones, as labels often cannot be obtained for large problem instances due to computational cost.

As an alternative to learning to imitate a (possibly sub-optimal heuristic) function, the model can learn from itself via reinforcement learning. In order to frame training as such a reinforcement learning problem, we need to express the task as a Markov decision process (MDP), where in each given state, an action can be selected that leads to the next state and an observed reward for the state transition. The model's parameters are optimized to maximize observed rewards. This allows for learning new policies that are not limited by the performance of an imitated heuristic. However, reinforcement learning is notoriously tricky to get to work.

Another option is to train the model in an unsupervised fashion. If we can formulate a differentiable loss function that accurately measures the quality of a prediction while not requiring any label information, we can train the model by minimizing said loss function. This method is very practical but might lead to learning a solution for a linear relaxation of the problem.

Generalization. Obtaining a model that generalizes well, providing good results for all possible input graphs is unrealistic. For $k \geq 3$, it is even proven to be impossible to build a polynomial time algorithm that approximates the optimal balanced partitioning up to a constant factor for all input graphs [1]. We thus do not consider the worst possible inputs and instead focus on randomly generated graphs that are similar to real-world networks. Building models that generalize across graphs of varying density and size still poses a significant challenge.

Time Complexity. In order for an algorithm to be practical for even large graphs, its asymptotic runtime must not exceed $\mathcal{O}(M)$ or at least $\mathcal{O}(N^2)$, where N and M are the number of nodes and edges, respectively. This rules out many potential algorithm designs.

1.2 Contributions

We present two approaches for building algorithms for balanced graph partitioning using learned heuristics. The first approach relies entirely on an end-to-end machine learning model. The model directly predicts the partition assignment of each node in a classification-like manner, cutting out the meta algorithm entirely. We train it in an unsupervised manner using a loss function that measures the quality of a given partitioning. The second approach is based on a meta algorithm that iteratively improves a given partitioning by swapping nodes' partition assignments. These nodes are chosen heuristically according to predictions of a machine learning model that is trained via Deep Q-learning. To this end, we present a useful trick, separating the Q-functions into two parts in order to reduce the asymptotic complexity. We further propose a new model architecture for generating node embeddings that capture structural information even when only synthetic node features are available. This proves to be useful for both presented approaches.

In the remainder of this thesis, we first provide necessary background information about the problem at hand and the methods we use. We then provide a brief overview over related work in the field of machine learning for combinatorial optimization. Further, we describe both the end-to-end approach and the iterative improvement approach in detail and finally present the results we obtain.

Chapter 2

Background

In the following, we provide the required background knowledge in the most relevant fields. First, we formally define the balanced graph partitioning problem. We then provide an overview of graph partitioning methods that are extended in this paper or used as a baseline. Further, we provide an overview of machine learning on graph structured data, including graph convolutional networks; a model architecture that both of the presented approaches rely on. Finally, we explain the most important concepts of reinforcement learning that we use for the iterative improvement approach.

2.1 Problem Definition

Graphs are a very general abstract concept to express binary relationships between arbitrary objects. The set of objects is represented by a set of nodes, where two nodes are connected by an edge whenever the respective objects are related. Formally we can define:

Definition 2.1 (Graph) *An (undirected, weighted) graph G is a tuple $(\mathcal{V}, \mathcal{E}, w)$, where*

- *$\mathcal{V}$ is the set of nodes (also called vertices)*
- *$\mathcal{E} \subseteq \{\{u, v\} \mid (u, v) \in \mathcal{V} \times \mathcal{V}$ and $u \neq v\}$ is the set of edges*
- *$w : \mathcal{E} \to \mathbb{R}_{>0}$ is the cost function, associating a positive weight with each edge*

We generally use the following conventional notation:

- *N and M are the number of nodes $|\mathcal{V}|$ and edges $|\mathcal{E}|$, respectively*
- *$\mathcal{N}(u)$ is the neighborhood $\{v \mid \{u, v\} \in \mathcal{E}\}$ of node u*
- *$\deg(u)$ is the degree $|\mathcal{N}(u)|$ of node u*

This concept is very useful, as a variety of problem inputs can be represented abstractly as a graph. Such problems are then oftentimes equivalent to a well known graph problem and can be solved much more easily due to the rich literature on graph algorithms.

One such graph problem that arises e.g. in parallel computing [22] and VLSI design [26] is to partition a graph into multiple pieces that are roughly equal in size and have a low number of edges connecting them:

Definition 2.2 (Partitioning) *A k-partitioning $\Pi_k = \{V_1, \ldots, V_k\}$ of a graph $G = (\mathcal{V}, \mathcal{E}, w)$ divides its set of nodes into k non-empty disjoint subsets $V_i \subseteq \mathcal{V}$ such that $\bigcup_{i=1}^{k} V_i = \mathcal{V}$.*

We use the notation $\Pi_k(v)$ for denoting the partition V_i that nodes node v belongs to in the partitioning Π_k, overloading the Π symbol.

Definition 2.3 (Cut Weight) *The cut weight of partitioning Π_k for a graph $G = (\mathcal{V}, \mathcal{E}, w)$ is*

$$\mathrm{cw}(\Pi_k) = \sum_{\substack{\{u,v\} \in \mathcal{E} \\ \text{with } \Pi_k(u) \neq \Pi_k(v)}} w(\{u, v\})$$

Definition 2.4 (Imbalance) *The imbalance of partitioning Π_k for a graph $G = (\mathcal{V}, \mathcal{E}, w)$ is*

$$\mathrm{imb}(\Pi_k) = \max_{V \in \Pi_k} \quad \min \quad \left\{ \varepsilon \in \mathbb{R}_{\geq 0} \mid |V| \leq (1 + \varepsilon) \cdot \left\lceil \frac{n}{k} \right\rceil \right\}$$

Formally, given a graph G, a number of partitions k, and imbalance factor ε, the ε-balanced graph partitioning problem is to find a partitioning Π_k of G with minimal cut weight satisfying the imbalance constraint:

$$\underset{\substack{\Pi_k \\ \text{with } \mathrm{imb}(\Pi_k) \leq \varepsilon}}{\operatorname{argmin}} \quad \mathrm{cw}(\Pi_k)$$

Other problem formulations incorporate node weights, hypergraphs, or different objectives [45, 7]. However, these alternative formulations are beyond the scope of this thesis. We also exclusively consider unweighted graphs with $\forall e \in \mathcal{E} \; w(e) = 1$, even if our methods could be extended to apply to weighted graphs.

2.1.1 Time Complexity

The unbalanced version of the problem (i.e. $\varepsilon = \infty$) with $k = 2$ is known as the min-cut problem. It can be solved deterministically in time $\mathcal{O}(MN + N^2 \log N)$ [54] or in $\mathcal{O}(M \log^3 N)$ with high probability using a Monte Carlo algorithm [25]. However, the problem complexity increases significantly with bigger k. Taking k as an input makes the problem NP-hard [17] and the best known deterministic algorithm has runtime $\tilde{\mathcal{O}}(N^{2k})$ [57].

In this work however, we focus on the balanced case. Requiring a balancing constraint ε makes the problem even more complex. The balanced bipartitioning problem is already NP-hard and so is balanced k-partitioning for bigger k [16, 24]. For the perfectly balanced case with $\varepsilon = 0$, it has been shown that, when $k \geq 3$, the optimal solution cannot be approximated up to a finite constant factor in polynomial time unless $P = NP$ [1].

2.2 Problem Relevance

Partitioning a graph into similar-sized pieces with few connections is a problem that arises in many applications including parallel processing, computer vision, and VLSI Design.

In parallel processing, graph partitioning is typically applied to ensure load balancing and minimize required communication [22]. To achieve maximal performance, each machine should process a similar workload and tasks dependent on each other's results should be assigned to the same machine as network communication is expensive. This optimization problem boils down to balanced graph partitioning where the tasks represent nodes that are connected by an edge whenever they require communication. When running algorithms on large graphs in a distributed setting, it is useful to partition the input graph itself onto different machines. This is particularly relevant for social media and internet companies that regularly deal with large networks.

In computer vision, graph partitioning is particularly relevant for image segmentation [48]. Given an image with a similarity metric between pixels, segmenting the image to disambiguate different objects can be done by partitioning the image into subsets of pixels representing the objects. Two pixels are connected by an edge if their similarity exceeds a given threshold. Alternatively, all pixels are connected with edges weighted according to a similarity metric.

Designing circuits for VLSI (Very Large Scale Integration) systems requires partitioning circuits into subcircuits while minimizing the wire length between connected circuits. This problem lends itself to applying Graph Partition algorithms [26]. In fact, the Kernighan-Lin algorithm [29], one of the first graph partitioning algorithms, was developed for optimizing component placement on circuit boards.

Further applications include Computer Aided Design (CAD), route planning, sparse matrix decomposition, and mapping of DNA [12, 7].

2.3 Graph Partitioning Algorithms

We provide a brief overview of methods for graph partitioning with a focus on the K/L algorithm which we adapt to use learned heuristics and on multilevel methods that the current state of the art solvers rely on. For more information, we refer to the survey papers on graph partitioning by Buluç et al. and Elsner [7, 12].

2.3.1 Kernighan-Lin Algorithm

The Kernighan-Lin algorithm (K/L) [29] for balanced bipartitioning takes an existing partitioning and improves it by repeatedly swapping pairs of nodes that are in different partitions to decrease the cut weight. This is done until there is no sequence of greedily chosen swaps that improves the partitioning.

Given a bipartitioning $\Pi_2 = \{V_1, V_2\}$ for a graph $G = (\mathcal{V}, \mathcal{E}, w)$, let the diff value D_v be the decrease in cut size when moving node v to the other partition; i.e. assuming w.l.o.g. $v \in V_1$ then

$$D_v = \sum_{u \in V_2 \cap \mathcal{N}(v)} w(\{u,v\}) - \sum_{u \in V_1 \cap \mathcal{N}(v)} w(\{u,v\}).$$

When swapping two nodes $v \in V_1$ and $u \in V_2$ (i.e. moving v to V_2 and u to V_1), we can now efficiently calculate the resulting decrease in cut size $\text{gain}(u,v)$ efficiently given their diff values D:

$$\text{gain}(u,v) = D_u + D_v - \begin{cases} 2 \cdot w(\{u,v\}) & \text{if } \{u,v\} \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases}$$

Using these concepts, the Kernighan-Lin algorithm improves an initial partitioning $\Pi_2 = \{V_1, V_2\}$ as described in Algorithm 1. Such an initial partitioning may be a random balanced partitioning, or a partitioning generated by a different partitioning method that we aim to improve.

The asymptotic complexity of this algorithm as originally designed is $\mathcal{O}(n^2 \log n)$ but several improvements and modifications with complexities down to $\mathcal{O}(m)$ have been proposed [11, 27, 15].

We are using the concept of iteratively improving a partitioning by swapping nodes to build a machine learning based algorithm. As a baseline, we also consider a greedy variant of the K/L algorithm that executes the swap with biggest gain immediately and terminates as soon as there is no single swap that improves the cut weight.

Algorithm 1: Kernighan-Lin Algorithm (as described by Elsner [12])

```
Compute $D_v$ for all vertices $v \in \mathcal{V}$
$A \leftarrow V_1,\ B \leftarrow V_2$
$s_0 \leftarrow 0$
for $i \leftarrow 1$ to $\min(|V_1|, |V_2|)$ do
    find $(a_i, b_i) \in A \times B$ such that $\text{gain}(a_i, b_i)$ maximal
    remove $a_i$ and $b_i$ from $A$ and $B$ respectively
    for $v \in \mathcal{N}(a_i) \cup \mathcal{N}(b_i)$ do
        // update $D_v$ as if $a_i$ and $b_i$ had been swapped
        $D_v \leftarrow D_v + 2 \cdot \begin{cases} w(\{v, a_i\}) - w(\{v, b_i\}) & \text{if } v \in V_1 \\ w(\{v, b_i\}) - w(\{v, a_i\}) & \text{if } v \in V_2 \end{cases}$
        // compute cut size as if $a_{1:i}$ and $b_{1:i}$ were swapped
        $s_i \leftarrow s_{i-1} - \text{gain}(a_i, b_i)$
    end
end
$j \leftarrow \underset{i}{\text{argmin}}\ s_i$
// swap the first $j$ pairs
$V_1 \leftarrow V_1 \backslash \{a_1, \ldots, a_j\} \cup \{b_1, \ldots, b_j\}$
$V_2 \leftarrow V_2 \backslash \{b_1, \ldots, b_j\} \cup \{a_1, \ldots, a_j\}$
repeat until no further cut-size improvement is achieved
```

2.3.2 Multilevel Methods

Many of the current state of the art solvers employ a multilevel approach. This means that they repeatedly coarsen (i.e. shrink) the graph, often by contracting edges. They then find a good (initial) partitioning of the smaller graph. In the uncoarsening stage, this partitioning is then mapped to the bigger, finer graphs where it is refined at each level to improve its quality given the additional information.

In this way, the algorithm performs both, more global and more local optimizations. When generating the initial partition on the coarsened graphs, the global structure is more relevant. However, when refining the graphs in the uncoarsening stage, the partitioning is improved locally. Furthermore, the trade-off between runtime and partition quality can be controlled well by deciding how much time is spent refining partitions and how much to coarsen the graph.

For coarsening a graph, edges or groups of nodes have to be contracted. Many heuristics have been proposed for selecting such edges / groups of nodes. Some of them are e.g. based on finding a matching with particularly heavy edges which are unlikely to be cut by a good partitioning and can thus relatively likely be contracted without negative consequences [7].

The initial partitioning of the small, coarsened graph forms the basis of the final partitioning. This initial partitioning is usually generated by one or several slower but precise algorithms, as runtime is not a constraint when partitioning such smaller graphs but high quality is important.

In the uncoarsening phase, existing partitions are improved. This is usually done with iterative improvement algorithms like K/L or it's variations.

We compare our methods against the state of the art graph partitioner hMETIS [28] that employs such multilevel partitioning schemes and produces partitions of very high quality. Other notable graph partitioning frameworks that rely on multilevel methods include Jostle, Scotch, and KaHIP [60, 47, 50].

2.3.3 Other Methods

A variety of other methods have been proposed for graph partitioning. We provide a brief overview over approaches that are not used in this thesis.

Graph growing methods start at some randomly or heuristically selected starting vertices and incrementally build partitions by adding boundary nodes. This can be done via simple breadth-first search (BFS) or by employing heuristics [14]. Such methods methods are very fast but do not always generate partitionings of high quality.

Spectral partitioning methods perform balanced partitioning with a more global view by utilizing properties of the second Eigenvalue and the corresponding eigenvector of the graph Laplacian matrix.

Other approaches include evolutionary algorithms [31], geometric methods, when node coordinates are available, and more described in the mentioned survey papers [7, 12].

2.4 Graph Machine Learning

In many real-world problems, the data has some underlying structure that can be characterized by a graph. When training machine learning models for such data it is crucial to utilize the information contained in the graph structure. In recent years, a lot of research has been directed at developing methods that do this effectively. We are dealing with featureless graphs i.e. only the structure is relevant and there is no data. Still, we can make use of the machine learning methods originally proposed for graph structured data. In the following, we provide a brief overview of the most relevant methods. For survey papers on geometric deep learning and graph neural networks, see the works by Bronstein et al. [5] and Wu et al. [64].

Earlier approaches to solve tasks on graph structured data make use of a loss function that is adapted to the graph structure and smoothes labels across neighboring nodes. Alternatively, solving problems on graph-structured data can be divided into two stages: First generating node embeddings that capture structural information about each node's neighborhood, then using these embeddings as input features for the downstream task [49, 18, 66]. Most of these methods are inherently transductive and do not generalize to unseen graphs.

2.4.1 Graph Convolutional Networks

The previously mentioned methods adapt the training technique or loss function to the graph structure. Graph Convolutional Networks (GCN), however, are models that are adapted to the graph structure, making them more adjusted to the task at hand. Different from standard Convolutional Neural Networks that are often used in Computer Vision, GCNs define Convolutions on signals on a graph structure i.e. node features, instead of signals in space or time. Zhang et al. provide a comprehensive review of GCNs [67].

Graph Convolutions can be defined in the spectral or "spatial" domain. Spectral convolution of a filter θ with a signal x on a graph can be performed through multiplication in the Fourier domain. The required graph Fourier transform is defined as multiplication with the Eigenvectors of the symmetric normalized graph Laplacian matrix. This concept can be used to define spectral graph convolutional networks with learnable filters [6, 21]. Defferard et al. reduce computational complexity of such networks by approximating the convolution operation using Chebyshev polynomials [10].

Derived from the Chebyshev approximation, Kipf and Welling introduce a method for graph convolution in the spatial domain [33]. Essentially, for each node, their GCN layer aggregates the features of neighboring nodes by addition and normalization and then applies a linear layer to generate the output embedding: Given an $N \times d^{(l)}$ feature matrix $H^{(l)}$, a GCN layer generates output embedding

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

where σ is a non-linearity like ReLU, $W^{(l)}$ is a learnable weight matrix, $\tilde{A} = A + I$ is the adjacency matrix with added self connections, and $\tilde{D}_{i,i} = \sum_{j=0}^{N} \tilde{A}_{i,j}$ the degree matrix of $\tilde{A}$. In order to look beyond the 1-neighborhood of each node, multiple such layers can be stacked; a model with k layers generates node embeddings based on nodes' k-neighborhood.

This model architecture outperforms all of the aforementioned methods on graph machine learning tasks commonly used as benchmarks. Additionally, it can be applied in the inductive setting.

Many alternative GCN formulations have been suggested [67]. Hamilton et al. propose GraphSAGE [20], where they introduce a variety of aggregator functions for computing a node's embedding based on its neighbors' embeddings instead of simply averaging them. They do not use self connections in the adjacency matrix. Instead they suggest to concatenate each node's representation with the aggregated representations of each neighbors for better gradient flow. In each layer l the representation $h_v^{(l)}$ of each node v is computed as

$$\begin{aligned} h_{\mathcal{N}(v)}^{(l)} &\leftarrow \text{aggregate}_l \left(\left\{ h_u^{(l-1)} \mid u \in \mathcal{N}(v) \right\} \right) \\ h_v^{(l)} &\leftarrow \sigma \left(W^{(l)} \cdot \text{concat} \left(h_{\mathcal{N}(v)}, h_v^{(l-1)} \right) \right) \\ h_v^{(l)} &\leftarrow h_v^{(l)} / \| h_v^{(l)} \|_2 \end{aligned}$$

with trainable weights $W^{(l)}$, non-linearity σ, input features $h_v^{(0)}$, and differentiable aggregator function aggregate$_l$. They suggest using the mean, max-pooling or LSTMs for aggregation. In order to reduce computational cost, they further propose to sample only a fixed number of neighbors for each node instead of considering all neighbors.

2.5 Reinforcement Learning

Reinforcement learning is a branch of machine learning concerned with optimizing the policy of an agent that acts in an environment and observes rewards for its actions. Reinforcement learning has been successfully applied on games like simple Atari computer games, chess, and Go [42, 52, 53]. More recently, it has also been applied to learn heuristics for combinatorial optimization problems [3, 43, 35, 9, 40]. We provide some background on the concepts that are relevant for this thesis. For further information we refer to the book on reinforcement learning by Sutton and Barto [55].

2.5.1 Markov Decision Process

The environment the agent acts in is usually described by a Markov decision process (MDP). Formally, we define an MDP as a 4-tuple

$$(S, A, \delta, \mathcal{R}),$$

where S is the state space, a set of possible states, A is the action space, a set of actions an agent can take, $\delta : S \times A \rightarrow S$ it the transition function that defines which state is reached after executing an action, and $\mathcal{R} : S \times A \rightarrow \mathbb{R}$ is the reward function that defines the reward observed when executing an action in a state. Here, we do not treat randomness in state transitions and consider δ and $\mathcal{R}$ to be deterministic.

Given an MDP, we define a policy as a mapping $\pi : S \rightarrow A$ that defines which action should be taken in each state. One is usually interested in finding a policy that maximizes the cumulative reward of all future actions. As this cumulative reward might be divergent when observing infinitely many rewards and as it is oftentimes more useful to obtain a reward soon than at some future point, it is useful to introduce a discount factor $\lambda \in [0, 1]$. We can then define the lifetime reward when executing policy π from starting state s_0 as

$$\sum_{i=0}^{\infty} \lambda^i \mathcal{R}(s_i, \pi(s_i))$$

with $s_i = \delta(s_{i-1}, \pi(s_{i-1}))$ for $i > 0$.

2.5.2 Deep Q-Learning

The goal of Q-learning is to learn a function $Q : S \times A \rightarrow \mathbb{R}$ that approximates the optimal action-value function

$$Q^*(s_0, a) = \mathcal{R}(s_0, a) + \max_{\pi} \sum_{i=1}^{\infty} \lambda^i \mathcal{R}(s_i, \pi(s_i)).$$

I.e. $Q^*(s, a)$ is defined as the lifetime reward when executing an optimal policy starting in state s and taking action a at first. Given such an approximation Q, a good policy can be constructed as

$$\pi(s) = \underset{a}{\operatorname{argmax}} \; Q(s, a).$$

In the case where $|S|$ and $|A|$ are small and finite, Q^* can be obtained by value iteration if the MDP is finite, or approximated in a table via standard Q-learning. However, when dealing with a big or infinite state space or action space, this becomes practically infeasible. A way to address this problem is to approximate the Q-function with a neural network $\hat{Q}$ as proposed by Mnih et al. [41]. In the classic deep Q-learning approach, experience is generated via executing an ε-greedy policy to generate experience (i.e. taking a random action with probability ε and a greedy action otherwise) and training the network based on this experience. Given an observed experience (s_i, a_i, r_i, s_{i+1}) of state, action, observed reward, and next state, the loss is computed as the mean squared error of the current $\hat{Q}$ prediction and the target value t_i

$$\mathcal{L}(s_i, a_i, r_i, s_{i+1}) = \left(\hat{Q}(s_i, a_i) - t_i\right)^2$$

where t is an improved estimate obtained via the Bellman equation

$$t_i = r_i + \lambda \cdot \max_{a'} \hat{Q}(s_{i+1}, a')$$

or $t_i = 0$ if s_{i+1} is a final state. This results in Algorithm 2.

Algorithm 2: Simple Deep Q-Learning

```
$\hat{Q} \leftarrow$ Neural Network
$\varepsilon \leftarrow$ exploration parameter $\in [0, 1]$
for $e \leftarrow 1$ to #episodes do
    $s \leftarrow$ starting state
    while $s$ not final state do
        $a \leftarrow \begin{cases} \text{random available action} & \text{with probability } \varepsilon \\ \operatorname{argmax}_a \hat{Q}(s, a) & \text{otherwise} \end{cases}$
        execute $a$ and observe reward $r$ and next state $s'$
        update $\hat{Q}$ parameters according to $\mathcal{L}(s, a, r, s')$
        $s \leftarrow s'$
    end
end
```

Many extensions and variations of this algorithm have been proposed. Mnih et al. [42] empirically show that introducing a replay memory and a target network stabilizes the learning process and enables significantly better performance. When using a replay memory, observed experience is stored in the replay memory and, instead of the newly observed experience, the training steps are performed on mini-batches sampled from the replay memory. This reduces correlations between observations and subsequent observations. When using a target network $\hat{Q}_{\text{target}}$, the target values are computed as $r_i + \lambda \cdot \max_{a'} \hat{Q}_{\text{target}}(s_{i+1}, a')$. The target network is synchronized with the training network only periodically, which reduces correlations between training predictions and target values.

To reduce short-sightedness, n-step updates are oftentimes used when applying Deep Q-Learning to combinatorial optimization [9, 40]. This means that the target is computed as

$$\sum_{i=0}^{n-1} \mathcal{R}(s_i, a_i) + \lambda \cdot \max_{a'} \hat{Q}(s_{i+n}, a'),$$

based on n experiences instead of a single one [55].

Another popular alternative to Deep Q-Leaning for training neural networks in the reinforcement learning setting are policy gradient methods [62, 56, 51]. These have also been been employed for combinatorial optimization [3, 43, 35]. However, in this work we focus on Deep Q-Learning.

Chapter 3

Related Work

Most algorithms that are used in practice for solving hard combinatorial optimization problems rely on hand-crafted heuristics. Recently, much research has been focused on developing machine learning models and training techniques for learning such heuristics instead. Here, we outline the most relevant work. For a more complete overview, we refer to the survey paper by Bengio et al. [4].

The traveling salesman problem (TSP) and its variations are the most studied problems in this domain. Vinyals et al. take a supervised approach based on a new model architecture, the pointer network which outputs a permutation of an input sequence [59]. They train the model using precomputed optimal solutions for graphs with up to 20 nodes and solutions computed by non-optimal algorithms for graphs with up to 50 nodes. Predictions are then generated via beam search. With this method, they achieve good results on problems that are not NP-hard and on small instances of the Euclidean TSP. However, the model does not generalize well to problem instances that are significantly bigger than those it was trained on.

Instead of supervised training on precomputed optimal solutions, Bello et al. [3] use policy gradient based reinforcement learning to train the pointer network without optimal solution labels. Their models achieve sightly better results on graphs with up to 100 nodes. Employing model ensembles, sampling multiple solutions, or fine-tuning the model to each problem instance during inference all yield further performance increases. Nazari et al. [43] apply this method to the more general vehicle routing problem using an adapted pointer network architecture.

Kool et al. [35] propose a Transformer [58] based model for solving routing problems. They train a Transformer Encoder and an attention based sequential decoder using a similar policy gradient based reinforcement learning method. They apply their approach to multiple routing problems, including the Euclidean TSP where they achieve improvements over the previously mentioned methods. However, they also only consider graphs with up to 100 nodes.

All of the aforementioned methods make use of node features, e.g. in the form of coordinates which are available in the Euclidean TSP. However, these are not typically available in all graph problems. Especially when creating solvers independent of a particular application, one cannot rely on node features; for different applications, the node features might have an entirely different meaning or there may not even be node features available. We consider the general case of abstract featureless graphs, i.e. we are not dealing with graph-structured data but with pure graph structures. Multiple methods have been proposed for learning heuristics on featureless graphs. Similar to our approach, most of them make use of existing model architectures for graph-structured data (mostly GCNs) and define synthetic node features.

Dai et al. [30] use structure2vec [9] model architecture similar to GCNs and simply provide zeros when no node features are available. They apply their approach to minimum vertex cover (MVC), maximum cut (MC), and the Euclidean TSP, that are all NP-hard. Comparable to our iterative improvement approach, they train their model to predict Q-values via n-step deep Q-learning. The solution is constructed incrementally by greedily adding the node with the highest predicted Q-value of the remaining nodes. The reward is defined as the increase in solution quality after adding a node. This means that rewards are observed much more frequently (i.e. at each meta algorithm step) compared to the previously mentioned reinforcement learning approaches where only the complete solution is rewarded. For MVC and MC, their method achieves only slightly worse results than the CPLEX MILP solver with a one hour cutoff on randomly generated Barabási-Albert [2] and Erdős–Rényi [13] graphs. It generalizes well from smaller graphs to graphs with up to 1200 nodes. On real world graphs, the method clearly outperforms simple baselines for all problems.

Li et al. [39] use supervised learning on optimal solutions for solving the NP-hard maximal independent set (MIS) problem. They train a graph convolutional network with ones as input features to predict probability maps over the vertices. Given a prediction, they greedily generate a partial solution and recursively apply the method on the residual graph. Further, they train the model to predict multiple alternative solutions at each step and perform a tree search to find a good sequence of predictions. The method achieves good results on MIS problems and problems that can be reduced to MIS. It is competitive with general MILP solvers and a state of the art MIS solver and generalizes well to graphs with up to 100,000 nodes. However, while the learned heuristics seem effective, the approach is also heavily relying on algorithmic search. This leads to frequent evaluations of the GCN which are expensive for larger graphs.

Mittal et al. [40] also use synthetic node features with a GCN-style model. However, they propose a two stage approach, first generating node embeddings that later serve as input for the downstream task of solving the influence maximization problem. The embeddings are generated by a GraphSAGE [20] model. In a supervised manner, this embedding model is trained to predict the marginal gain contributed by each node to the solution, averaged over a number of greedy solutions. As this proxy task is related to the downstream task, the model is expected to extract useful features. A solution is then incrementally constructed by adding nodes according to decisions based on an n-step Q-Learning approach similar to our iterative improvement approach or the method suggested by Dai et al. [30]. However, the current state is encoded by max pooling over the embeddings of the nodes in the current solution instead of encoding it as input features of the embedding model as proposed by Dai et al. This allows for computing the expensive node embeddings only once, yielding remarkably fast performance. The authors report runtimes of under a second on graphs with 500,000 nodes and significant speedups over the algorithmic state of the art solver and a greedy algorithm. However, in terms of solution quality, the proposed method is consistently worse than the state of the art solver and even inferior to the greedy approach when applied to the similar Maximum Vertex Cover problem.

All of these methods construct solutions by incrementally adding nodes to a solution set. They are designed for problems where a solution can be characterized as a set of nodes. This is not the case for balanced graph partitioning (at least not for $k > 2$). There, a solution consists of multiple sets of nodes, or a label on each node that assigns it to a partition. Hence, the aforementioned methods cannot naturally be applied to balanced graph partitioning.

To our knowledge, the only attempt to use machine learning for balanced graph partitioning is the GAP framework [44] after which our end-to-end approach is oriented. Instead of supervised learning or reinforcement learning, the authors use unsupervised learning with a loss function that characterizes solution quality to train a model to immediately generate a partitioning. They use an embedding model (GCN or GraphSAGE) followed by an MLP with softmax activation that generates a output vector of length k for each node, representing the probability of the node pertaining to each respective partition. The loss is then computed as the sum of the normalized expected cut weight and the mean squared error (MSE) between the expected partition size and the ideal partition size. When training and evaluating one instance at a time, this method produces better partitionings than the state of the art partitioner hMETIS [28] on graphs with node features (data flow graphs of neural networks). They also report generalization ability; when training the model on random graphs and evaluating on a different test set, the results are on par or minimally better compared to hMETIS. However, as these random graphs do not have features, the authors suggest to perform principal component analysis (PCA) on the graph adjacency matrix and to use the extracted vectors as node features. With an $\mathcal{O}(N^3)$ runtime, this is prohibitively expensive for large graphs. We propose methods that do not rely on node features.

Chapter 4

Data

For both training our models and evaluating the effectiveness of our methods, we need graph data. In order to be able to control parameters like graph size and density, we randomly generate graphs according to four commonly used models that produce graphs similar to real-world networks. We use implementations provided in the NetworkX Python library [19]. To avoid disconnected graphs we generate new random graphs until the obtained graph is connected.

4.1 Graph Generation Models

Erdős–Rényi Model

In the Erdős–Rényi model (ER) [13], the number of nodes n is fixed and each edge is created independently and equally likely with probability p. This leads to an expected number of edges of $p \cdot \binom{n}{2}$. The resulting graphs have low clustering and are likely to be connected if $n \cdot p > 1$ [13].

Watts–Strogatz Model

Graphs generated following the Watts-Strogatz model (WS) are so called small-world graphs with short path lengths and high local clustering [61]. These properties are also common in real-world social networks. A Watts-Strogatz graph with n nodes is generated by forming a ring and connecting each node with its m nearest neighbors in this ring. Finally, for each node, each adjacent edge is reconnected to a random node with probability p. We observe that the graphs allow for cheaper partitionings when increasing this randomness parameter. For $p = 0$ the graph has a very regular structure whereas the random reconnections enable opportunities for cheaper cuts.

Barabási–Albert Model

The Barabási-Albert (BA) model produces graphs where node degrees follow a scale-free power-law distribution, similar to real-world networks like the World Wide Web [2]. To generate a graph with n nodes, m initial nodes with degree 0 are created. The remaining nodes are added iteratively and each new node is connected to m existing nodes, where it is preferentially attached to nodes with high degree.

Random Geometric Graphs

To build random geometric graphs (GEO), n nodes are generated as points chosen uniformly at random from the unit cube of dimension d. Two nodes are connected by an edge if the corresponding points have distance less than the radius parameter r.

4.2 Experimental Comparison

We observe that, for all graph types, the weight of a random partitioning increases linearly with increasing graph density (see Figure 4.1). The weight of a partitioning computed by state of the art solver hMETIS increases linearly with respect to graph density for Erdős–Rényi and Barabási–Albert graphs. For both, Watts-Strogatz graphs and random geometric graphs, it seems to increase superlinearly. Graphs generated with the latter models also seem to be better partitionable; hMETIS is able to find partitionings that are significantly cheaper than random partitionings.

When choosing parameters such that the density is fixed, the cut weight of partitionings scales quadratically with respect to the number of nodes, as expected (see Figure 4.2). This is the case for both random balanced partitionings and partitionings computed by the hMETIS solver. However, the latter consistently produces better partitionings. Again, this difference is particularly pronounced for the Watts-Strogatz graphs and the random geometric graphs.

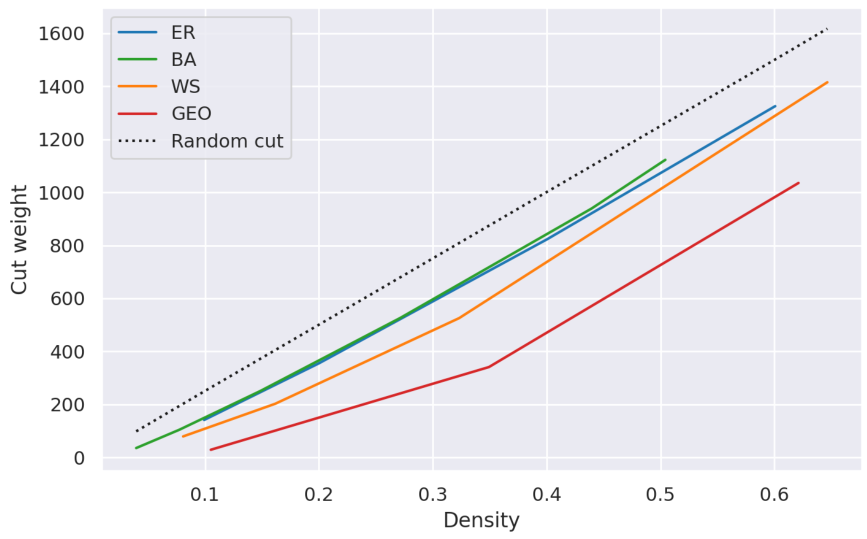


Figure 4.1: Partitioning cut weight by graph density (bi-partitioning). Partitionings computed by hMETIS with the lowest possible imbalance parameter. The average weight of a random balanced partitioning for a given density is equal across all graph types. Mean of results of five runs on 100 graphs of size 100. We compare graphs with the parameters from Table A.3 as well as BA graphs with $m \in \{2, 4\}$.

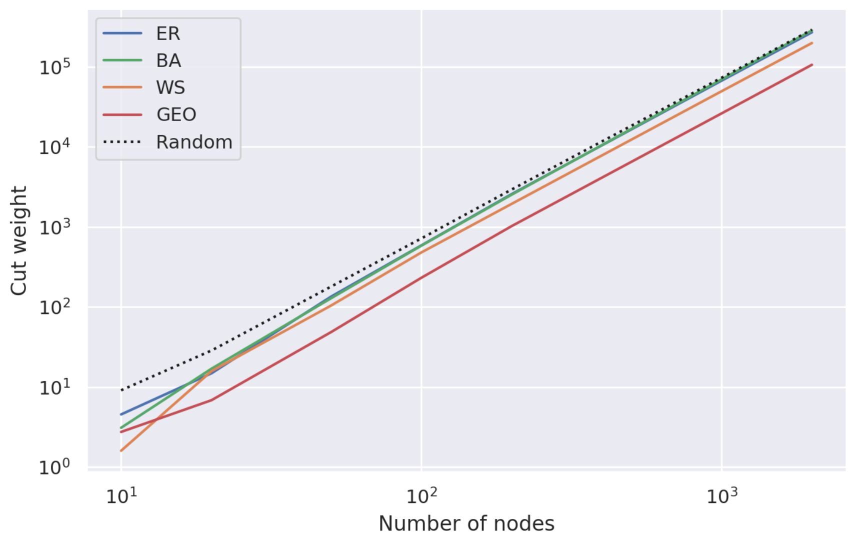


Figure 4.2: Partitioning cut weight by graph size (bi-partitioning). Partitionings computed by hMETIS with the lowest possible imbalance parameter. The average weight of a random balanced partitioning for a given size is equal across all graph types. Mean of results of five runs on 20 graphs per size / type with density ≈ 0.3. For the exact parameters, see Table A.2.

Chapter 5

Methods

We present two approaches for employing machine learning in algorithms for balanced graph partitioning. First we describe the end-to-end approach which is entirely based on a machine learning model and allows interesting insights into how useful pure machine learning is in this setting. Then, we describe the iterative improvement approach that is based on a node swapping meta algorithm and a neural network trained via reinforcement learning.

5.1 Node Embeddings

Both approaches rely on neural networks that operate on vectors in Euclidean space. In order to represent the graph structure in Euclidean space, we generate node embeddings that capture structural information. The embedding of a node is a vector that ideally contains information about its local neighborhood and its global role in the graph. Such vector representations are useful as input to model components that solve the downstream task and approximate heuristics.

5.1.1 Node Features

We use GCN-style models to generate such node embeddings. These have been shown to capture structural information of nodes' neighborhoods [33, 20]. However, they rely on input features. We are dealing with the general case of featureless graphs where only the graph structure is relevant and there is no underlying data. Similar to other work on combinatorial optimization problems [39, 40, 44], we therefore design synthetic input features. We propose three different ways of generating input features.

One The number 1 as the single feature.

Degree The number 1 and the degree of the node divided by N for normalization.

DegreeSorted The number 1, the degree of the node divided by N. Additionally, the normalized position ($\in [0,1]$) of the node in the sequence of all nodes sorted by degree with random tie breaking.

These features contain varying degrees of information. When using One features, the model is expected to compute all useful structural information. The Degree features contain local information about the size of a node's neighborhood, while the DegreeSorted features additionally contain global information. In order to ensure that the features vectors of the three different methods have the same length, we always use ones to pad each feature vector to length four.

5.1.2 Model Architecture

A GCN-style model architecture for this setting has to be expressive enough to extract useful information from the given features. This is not the case for all architectures. A commonly used architecture similar to a variant of GraphSAGE [20] aggregates over neighbors by computing the mean of their representations. We refer to this architecture as MeanGCN and describe it in detail in Appendix A.1. For a node u it aggregates the representations $h_v^{(l)}$ of neighboring nodes as

$$\mathrm{agg}_u^{(l)} = \frac{1}{\deg(u)} \sum_{v \in \mathcal{N}(u)} h_v^{(l)}$$

in layer l. This technique fails to extract any useful information when using One features: the result of the aggregation will be one, independently of a nodes' neighborhood. Even with the other features, the resulting aggregated information of two different nodes' neighborhoods is often very similar. This makes it hard for the model to extract structural information. Indeed, we observe this experimentally (see Section 6.1.1), even if Mittal et al. report good results with this method [40].

The architecture proposed by Kipf and Welling [33], henceforth SymGCN, employs symmetric normalization when aggregating (see also Appendix A.1):

$$\mathrm{agg}_u^{(l)} = \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{1}{\sqrt{\deg(v) \cdot \deg(u)}} \cdot h_v^{(l)}$$

This alleviates the issue slightly but it still fails to extract useful features (see Section 6.1.1). We observe that the standard deviation between the generated node embeddings is often very low.

GraphNorm GCN To combat this, we propose to aggregate by summing over the neighborhood and then, after multiplication with the learned weights, normalizing towards a mean of zero and a standard deviation of one across the embeddings of all nodes in the graph. We call this method GraphNormGCN. Algorithm 3 describes the resulting propagation rule in layer l, given input representations $h_v^{(l)}$ and learnable weight matrix $W^{(l)}$.

Algorithm 3: GraphNormGCN

```
$s_u^{(l)} \leftarrow \sum_{v \in \mathcal{N}(u)} h_v^{(l)}$
$x_u^{(l)} \leftarrow W^{(l)} \left( s_u^{(l)} || h_u^{(l)} \right)$ // concatenation
$\mu^{(l)} \leftarrow \frac{1}{N} \sum_{v=1}^{N} x_v$
$\sigma^{(l)} \leftarrow \sqrt{\frac{1}{N-1} \sum_{v=1}^{N} \left( x_v - \mu^{(l)} \right)^2}$
$\text{agg}_u^{(l)} \leftarrow \left( s_u - \mu^{(l)} \right) \oslash \sigma^{(l)}$ // element-wise division
$h_u^{(l+1)} = \text{elu}(\text{agg}_u^{(l)})$
```

We use the sample mean and the unbiased sample standard deviation. This normalization trick guarantees a greater variance among the node embeddings and makes differences in degree more pronounced. It also makes the embeddings of single nodes heavily dependent on the graph context. Two nodes that have the same neighborhood but belong to different graphs may have embeddings of entirely different scale. This also means that an embedding contains global information about the graph beyond its neighborhood, which is desirable for our use case. To our knowledge, we are the first to explore this method.

Max Pooling As an alternative to aggregating via mean or sum, we explore max pooling of nodes' neighbors' embeddings as proposed by Hamilton et al. [20]. This might enable the embedding model to pick up on extreme features of single nodes instead of a smoothed mean which could be particularly helpful in the combinatorial setting where small deviations of single nodes can be highly relevant. We evaluate two variants of GraphNormGCN with max pooling: GraphNormGCN+Pool where we replace the sum over the neighbors with max pooling and GraphNormGCN+Pool+Sum where the pooling acts as an additional method of aggregation and is concatenated to the sum.

5.2 End-to-end Approach

Instead of learning only heuristics, we explore an approach that does not rely on a meta algorithm but is entirely based on a learned model. We frame graph partitioning as a classification task where each node is assigned a partition. The model receives node features and the adjacency matrix as input and predicts the partition assignment of each node. We expect this approach to be less precise than methods that employ stronger meta algorithms. Yet, it provides insight into the effectiveness of different model architectures in this setting.

Nazi et al. have presented promising results with this method [44]. However, their approach relies on node features that are either given by the underlying data or generated expensively by PCA on the adjacency matrix. We use the cheap synthetic node features described in section 5.1.1.

5.2.1 Training

Machine learning models that solve such classification tasks are often trained in a supervised manner. In this setting, however, it is expensive, if not infeasible, to obtain ground truth labels of optimal partitionings. Additionally, there may be various optimal partitionings for a given graph, which would further complicate supervised training. Instead, as suggested by Nazi et al. [44], we opt for an unsupervised approach with a differentiable loss function that expresses the quality of a predicted partitioning. This loss function has to account for both objectives by penalizing imbalanced partitionings as well as expensive cuts.

The model outputs matrix $P \in \mathbb{R}^{N \times k}$, where $P_{i,p}$ is the predicted probability that node i is in partition p. Let Π_k be the random k-partitioning resulting when assigning each node's partition independently according to P.

Balance Loss We can compute the squared relative error between the expected weight of partition p and the ideal weight when balanced as

$$\mathrm{RE}_{\mathrm{imb}}^{p} = \left(1 - \frac{k}{N} \cdot \sum_{i=0}^{N} P_{i,p}\right)^2$$

and define a loss function as the mean of these errors over all partitions:

$$\mathcal{L}_{\mathrm{imb}} = \frac{1}{k} \sum_{j=1}^{k} \mathrm{RE}_{\mathrm{imb}}^{j}$$

This serves as a better objective than $\mathbb{E}[\mathrm{imb}(\Pi_k)]$ as it takes into account the weight of all partitions instead of only the heaviest. This loss also has comparable magnitude across various N and k in respect to the imbalance.

Cut Loss Given the adjacency matrix A we can additionally compute the expected cut weight. Based on

$$B = P \odot (A(1 - P))$$

where $B_{i,j}$ is the expected number of neighbors of node i that are not in partition j when i in partition j ($\odot$ denotes the element-wise Hadamard product), we can compute C with

$$C_p = \sum_{i=1}^{N} B_{i,p}$$

which is the expected number of edges from partition p into other partitions. We then obtain

$$\mathbb{E}[\mathrm{cw}(\Pi_k)] = \frac{1}{2} \sum_{j=1}^{k} C_j.$$

In order to obtain a loss with comparable magnitude for similar solution quality over different M, N, and k, we divide by the expected cut weight of a random partitioning. On an Erdős–Rényi graph in which each edge is independently equally likely, the expected weight of a random balanced partitioning is $M \cdot \frac{k-1}{k}$. We thus define

$$\mathcal{L}_{\text{cut}} = \frac{\mathbb{E}[\mathrm{cw}(\Pi_k)]}{M \cdot \frac{k-1}{k}} = \frac{k}{2M(k-1)} \sum_{j=1}^{k} C_j.$$

Overall Loss Accordingly, we define the overall loss function as the sum of the loss function for the balancing objective and the loss function for the cut weight objective. To ensure that the loss functions are of similar magnitude, we scale the imbalance loss by 100.

$$\mathcal{L} = 100 \cdot \mathcal{L}_{\text{imb}} + \mathcal{L}_{\text{cut}}$$

5.2.2 Model Architecture

We propose a model architecture with two components: A node embedding model consisting of a three layer GCN followed by a classification model with two fully connected layers and a Softmax activation that outputs partition predictions for each node (see Figure 5.1). Similar to the approach proposed by Nazi et al. [44], the partitioning is generated by independently assigning each node to its most likely partition. The resulting inference runtime is $\mathcal{O}(M)$ when using a sparse GCN implementation.

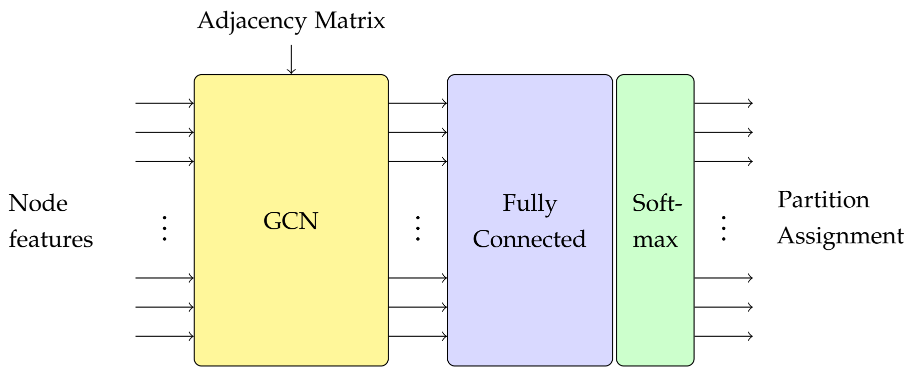


Figure 5.1: The base model architecture for the end-to-end partitioning approach

Node embeddings computed by the three layer GCN are only based on information about each node's 3-neighborhood. Hence, these embeddings cannot capture global structural information in a graph with diameter greater than 3. To overcome this limitation, we propose two extensions of the model.

Transformer Encoder We introduce a Transformer Encoder [58] before the linear layers as an additional model component Transformer. Specifically, we use a Pre-LN Transformer architecture. As pointed out by Xiong et al. [65], this architecture trains more robustly and we observe better gradient flow. The Transformer component allows each node to attend on all other nodes. We expect that this yields node embeddings that contain information about nodes' global roles. The quadratic Transformer Encoder increases the overall complexity to $\mathcal{O}(N^2)$.

Attention-based Graph Embedding As an alternative to the expensive Transformer Encoder, we propose the GraphEmbedder model component that computes a representation of the entire graph based on a linear attention mechanism. Similar to the Transformer Encoder, it is also based on scaled dot-product attention. However, instead of a self-attention approach where the embeddings are used as queries, keys, and values, the model generates only a constant number of queries (here 4). These are generated by three fully connected linear layers. In order to generate queries that are adapted to the graph at hand, the query generator receives N, M, and the graph density as input. It also receives the mean embedding across all nodes. This provides rough information about the position of the node embeddings in the embedding space. The multi-head scaled dot-product attention mechanism then produces a constant number of output vectors in $\mathcal{O}(N)$. In a final linear layer, these are combined into a graph embedding.

5.3 Iterative Improvement Approach

The end-to-end approach relies entirely on a machine learning model. Partitions are generated directly from the model output, implicitly assuming independence of different nodes' partition assignments, which is not the case. Due to the combinatorial nature of the problem, it is to be expected that such an approach cannot always generate close to optimal partitionings, let alone guarantee to satisfy a given balancing constraint. Using a meta algorithm that guides the solving process and makes decisions based on learned heuristics, it is possible to provide such guarantees. Furthermore, a carefully designed meta algorithm can exploit prior knowledge about the problem structure, making it easier to learn useful heuristics.

Existing algorithmic solvers often follow a multilevel approach, first coarsening the graph repeatedly, then partitioning it and refining the partitioning at each uncoarsening stage [7]. When coarsening, the solver heuristically selects edges to contract. This selection could be performed by a machine learning model. However, currently employed heuristics for edge selection are already very fast. A machine learning model might yield more accurate heuristics. But accuracy is not a priority here, as even inaccurate decisions can be corrected in the refinement stage. Furthermore, it is not immediately clear how a model can be trained to effectively select edges for contraction.

In the refinement stage, many solvers rely on an algorithm that iteratively improves partitionings by moving or swapping single nodes (see Algorithm 4). This is also done heuristically with algorithms such as K/L. The heuristic selection of nodes lends itself to being performed by machine learning models. We hence propose an iterative improvement meta algorithm based on node swapping with a heuristic selection function, inspired by K/L. We describe it in detail in Algorithm 5. This Algorithm always produces perfectly balanced partitionings. The node selection heuristic only has to reduce the cut weight.

Algorithm 4: Swapping two nodes' partition assginments

```
def swap($\Pi_k, u_1, u_2$):
    $P_1 \leftarrow (\Pi_k(u_1) \setminus u_1) \cup \{u_2\}$
    $P_2 \leftarrow (\Pi_k(u_2) \setminus u_2) \cup \{u_1\}$
    $\Pi'_k \leftarrow \{V \mid V \in \Pi_k \wedge \Pi_k(u_1) \neq V \neq \Pi_k(u_2)\} \cup \{P_1, P_2\}$ [1]
    return $\Pi'_k$
```

[1]Recall that we overload the Π symbol such that $\Pi_k(v)$ denotes the partition V_i that nodes node v belongs to in the partitioning Π_k.

Algorithm 5: Iterative Improvement Meta Algorithm

```
$\Pi_k \leftarrow$ random balanced $k$-partitioning
for $i \leftarrow 0$ to $\lceil \alpha \cdot N \rceil$ do
    $u_1, u_2, \text{pred}_{\text{next}} \leftarrow \text{select}(\Pi_k)$ ;            // where $\Pi_k(u_1) \neq \Pi_k(u_2)$
    if $\text{pred}_{\text{next}} > 0$ then
        $\Pi_k \leftarrow \text{swap}(\Pi_k, u_1, u_2)$
    end
    else
        break
    end
end
return best $\Pi_k$ found
```

5.3.1 Markov Decision Process

The algorithm can naturally be interpreted as a Markov Decision Process (MDP) where swapping a pair of nodes is an action that causes a state transition to a different partitioning and leads to a reduction of the cut weight which represents a reward.

Formally, we define the corresponding deterministic MDP as a 4-tuple $(S, A, \delta, \mathcal{R})$. The state space S is the set of all graphs and their respective k-partitionings:

$$S = \{(G, \Pi_k) \mid G \text{ is a graph and } \Pi_k \text{ a } k\text{-partitioning of G}\}$$

The available set of actions A in a state $s = (G, \Pi_k)$ is the set of node swaps:

$$A = \{\{u_1, u_2\} \mid \Pi_k(u_1) \neq \Pi_k(u_2) \wedge u_1, u_2 \in G\}$$

The deterministic state transitioning function δ is defined by the node swapping operation as

$$\begin{aligned} \delta((G, \Pi_k), \{u_1, u_2\}) &= (G, \Pi'_k) \\ \text{with } \Pi'_k(v) &= \begin{cases} (\Pi_k(u_1) \setminus u_1) \cup \{u_2\} & \text{for } v = u_1 \\ (\Pi_k(u_2) \setminus u_2) \cup \{u_1\} & \text{for } v = u_2 \\ \Pi_k(v) & \text{otherwise} \end{cases} \end{aligned}$$

and we define the reward function $\mathcal{R}$ as

$$\mathcal{R}((G, \Pi_k), a) = \frac{\text{cw}(\Pi_k) - \text{cw}(\Pi'_k)}{M/N}$$

if $\delta((G, \Pi_k), a) = (G, \Pi'_k)$. M and N denote number of edges and nodes, respectively, and serve as normalization of the rewards: For a good selection policy for bi-partitioning, the expected decrease in cut weight per step grows approximately linearly with respect to $\frac{M}{N}$, as the cut weight grows roughly linearly in M and the expected number of swaps required to reach a given partitioning is in $\Theta(N)$.

5.3.2 Deep Q-Learning

Given the formulation as an MDP, we can use reinforcement learning for finding a good selection policy. In particular, we apply Deep Q-Learning and train a neural network $\hat{Q}$ to learn a Q-function that approximates the optimum action-value function. Based on $\hat{Q}$, we use a greedy policy for selecting which nodes to swap (see Algorithm 6). Together with the described meta algorithm, this yields a complete graph partitioning algorithm.

Algorithm 6: Deep Q-Learning Selection

```
def select($G, \Pi_k$):
    $s \leftarrow (G, \Pi_k)$
    $u_1, u_2 \leftarrow \operatorname{argmax}_{a \in A} \hat{Q}(s, a)$
    $\Pi'_k \leftarrow \text{swap}(\Pi_k, u_1, u_2)$
    $s' \leftarrow (G, \Pi'_k)$
    $\text{pred}_{\text{next}} \leftarrow \max_{a' \in A} \hat{Q}(s', a')$
    return $u_1, u_2, \text{pred}_{\text{next}}$
```

5.3.3 Model Architecture

The Q-model receives a state in form of a graph and a partitioning as well as an action in form of a node as input. For the same state s, $\hat{Q}(s, a)$ is computed multiple times for different a when searching for the best action. As computation on graphs is expensive, it therefore makes sense to design the model in such a way that a representation of the state is computed only once and can be reused for multiple actions. The same principle applies to graph and partitioning: over the course of an algorithm run, the graph stays the same while the partitioning changes at every step.

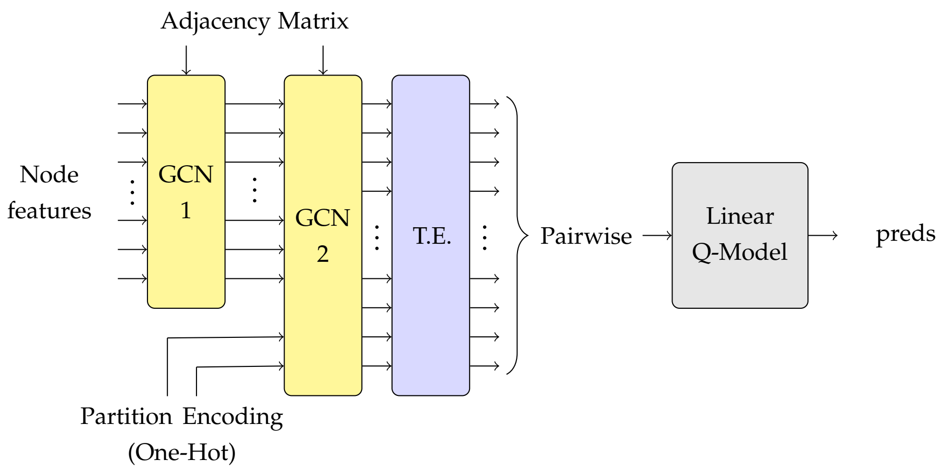


Figure 5.2: The base model architecture for the iterative improvement partitioning approach. T.E. denotes Transformer Encoder. The one-hot representation of each node's partition is concatenated with its embedding as input for GCN 2.

We therefore propose a modular design with three steps/components:

1. a three layer GraphNormGCN with DegreeSorted features that computes a representation of the graph in form of node embeddings
2. a single layer[2] GraphNormGCN with a Transformer Encoder that takes the node embeddings concatenated with the one-hot encoding of each node's current partition as input and produces node embeddings in the context of the full state including the partitioning
3. a fully connected feedforward network that predicts the Q-value of a swap given the representations of the two nodes

See Figure 5.2 for a detailed visualization.

The design choices are based on the results of the end-to-end approach. There, the GraphNormGCN proved most effective for generating node embeddings that contain information about the graph structure. Furthermore, an extra Transformer Encoder component helped in obtaining a more global view of the graph. In this setting, the Transformer Encoder also enables the transfer of information about the partition assignment across all nodes.

[2]When using only a single layer, the embeddings can be updated in $\mathcal{O}(N)$ after a node swap (see also Section 5.3.6 for runtime considerations).

5.3.4 Training

We train the model via Deep Q-Learning, generating experience by executing an ε-greedy policy (choosing a random action with probability ε) and as loss function using the mean squared error (MSE) between predictions and targets computed from the observed rewards via the Bellman equation (see Section 2.5.2). Additionally, we incorporate a replay memory, a target network, and we use n-step rewards. Reinforcement learning is notoriously tricky to get to work reliably and without these techniques we do not observe robust performance. We introduce them in detail in Section 2.5.2. Replay memory and target network are standard practice to increase robustness, while n-step rewards can help reduce short-sightedness. We also experience that n-step rewards enable us to train with bigger discount factors without observing overestimated Q-values (see Section 6.2.1).

With states s_i, rewards r_i, actions a_i, discount factor λ, and target Q-model $\hat{Q}^{\text{target}}$, we hence obtain

$$t_i = \sum_{j=i}^{i+n-1} r_j + \lambda \cdot \max_{a'} \hat{Q}^{\text{target}}(s_n, a')$$

as the target value at step i and compute the loss for the prediction of Q-model $\hat{Q}$ as

$$\mathcal{L} = \left(\hat{Q}(s_i, a_i) - t_i\right)^2.$$

In order to increase training speed, we do not make a training update after every observation. Instead, we only train periodically, but, when training, we sample many batches instead of a single one. This requires less expensive transitions between training and exploration and thereby yields faster runtimes. The resulting training process for a single graph is described in Algorithm 7. This algorithm is run on all graphs for a fixed number of epochs.

5.3.5 Finite MDP

In other work where reinforcement learning is applied to combinatorial optimization problems [40, 3, 9], the solution is often built incrementally, e.g. as a set of nodes. Using our iterative improvement approach, a solution is available at all times. The solutions are improved over time, but there is no clear notion of progress: If not limited by a maximum number of steps or terminated when no further improvement is predicted, the algorithm could theoretically continue swapping nodes indefinitely. From the perspective of

Algorithm 7: Deep Q-Learning Training Run

given: graph G, replay memory RM, networks $\hat{Q}, \hat{Q}^{\text{target}}$, step count t
parameters: reward steps n, train frequency f_{train}, update frequency f_{update}, discount factor λ, exploration ε

```
$\Pi_k \leftarrow$ random balanced $k$-partitioning
$s_0 \leftarrow (G, \Pi_k)$
for $i \leftarrow 0$ to $\lceil \alpha \cdot N \rceil$ do
    if explore with probability $\varepsilon$ then
        $u_1, u_2 \leftarrow$ random nodes from different partitions
    else
        $u_1, u_2, \text{pred}_{\text{next}} \leftarrow \text{select}(\Pi_k)$ ;    // selection using $\hat{Q}$
    $\Pi'_k \leftarrow \text{swap}(\Pi_k, u_1, u_2)$
    $r_i \leftarrow \frac{\text{cw}(\Pi_k) - \text{cw}(\Pi'_k)}{M/N}$
    $s_{i+1} \leftarrow (G, \Pi'_k)$
    if $i \geq n$ then
        // store in both ways as ordering has no meaning
        store $(s_{i-n}, (u_1, u_2), \sum_{j=i-n}^{i} r_j, s_i)$ in $RM$
        store $(s_{i-n}, (u_2, u_1), \sum_{j=i-n}^{i} r_j, s_i)$ in $RM$
    $t \leftarrow t + 1$
    if $t \mod f_{update} = 0$ then
        // update target network
        $\hat{Q}^{\text{target}} \leftarrow \hat{Q}$
    if $t \mod f_{train} = 0$ then
        // train
        sample batches from $RM$
        for sample $(s, a, r_{\text{cum}}, s')$ perform gradient step with respect to
            $\text{MSE}(\hat{Q}(s,a),\ r_{\text{cum}} + \lambda \cdot \max_{a'} \hat{Q}^{\text{target}}(s', a'))$
```

the Q-learning model, this is the case when training with Algorithm 7. Although only $\lceil \alpha \cdot N \rceil$ swaps are performed, the model does not receive this information as input. It is unaware of the step limit. Based on the information available to the model, the optimal swap can always be reached in the future because there are infinitely many remaining steps. Consequently, the Q-values of any swap are very high, irrespective of its actual benefit. Only the discount factor counteracts this effect. We observe this problem experimentally: When training with big discount factors, we observe overestimated Q-values (see Section 6.2.1). To overcome this problem, in step i we additionally input the relative progress $p = \frac{i}{\lceil \alpha N \rceil}$ as part of the current state: (G, Π_k, p). We then define the future rewards in all end states where $p = 1$ as zero: We define

$$\mathcal{R}((G, \Pi_k, 1), a) = 0$$

and set

$$\hat{Q}^{\text{target}}((G, \Pi_k, 1), a) = 0$$

for every G, a, Π_k. This adaption is necessary in order to correctly model the underlying finite MDP in the case that α is finite i.e. the number of steps are limited. It stabilizes the Q-values and helps incentivize the model to improve the partitioning before the step limit is reached.

5.3.6 Double Q-Function

For scaling the algorithm to very big graphs, the computation of the best predicted swap should ideally run in linear time, i.e. $\mathcal{O}(M)$. The base model architecture as described in Figure 5.2 does not satisfy this requirement. Both the GCN and the Transformer Encoder require $\mathcal{O}(N^2)$ time. However, with a sparse graph representation, the GCN can be implemented in $\mathcal{O}(M)$ and, while useful, the Transformer Encoder could be left out or replaced by a graph embedding model as described in Section 5.2.2. The actual bottleneck lies within the Q-learning formulation that requires finding the optimal pair of nodes to swap from $\mathcal{O}(N^2)$ candidate pairs. For each of these candidate pairs, the Q-model component has to be evaluated. This leads to an $\mathcal{O}(N^2)$ runtime, no matter how efficiently the model components operate.

To overcome this limitation, we propose an alternative problem formulation where the optimal node pair to swap is determined in two steps. First, one of the optimal candidates to partake in a swap is selected among all nodes in $\mathcal{O}(N)$ as the first node to be swapped. Then, given this first node, the second node is selected in $\mathcal{O}(N)$ as the best candidate to be swapped with the first node. This limits the overall runtime to $\mathcal{O}(N)$ and thus makes it possible to optimize the entire model to run in $O(M)$ or even faster [3] for

[3]The full GCN has to run only once in the beginning. After swapping two nodes, only them and their neighbors' representations have to be recomputed by GCN2. This can be

finding a single optimal swap. Here, we leave these optimizations for future work and show how the Q-learning approach can be modified.

MDP The new algorithm with a two step selection process can again be modeled as an MDP. The concept is very similar to the MDP described in Section 5.3.1. However, as there are two single node selection steps, an action is defined as the choice of a single node and the actions alternate between choosing a first node and choosing a second node. The choice of the first node does not yield a reward; the associated reward is 0. However, the choice of the first node has to be reflected in the state. The state is hence augmented to contain a selected node. When in a state in which a first node is selected, the next action corresponds to choosing a second node. All nodes in different partitions from the first node are available as actions. Once the second node is chosen, the swap is executed and reflected in the state transition. The associated reward is defined as the normalized reduction cut weight.

Deep Q-Learning We employ Deep Q-Learning as described earlier to learn useful policies. However, instead of a single Q-model, we use two Q-models $\hat{Q}_1$ and $\hat{Q}_2$ with shared weights. $\hat{Q}_1$ is used to select the first node and $\hat{Q}_2$ receives the first node as input and predicts the Q-values associated with choosing the remaining candidates for the next swap. This simplifies dealing with the different meaning of the alternating actions. The resulting model architecture is described in Figure 5.3. When executing a greedy policy using $\hat{Q}_1$ and $\hat{Q}_2$, we obtain Algorithm 8 for node selection. In practice, we consider the 8 top candidates for u_1 instead of just the single best candidate. This can compensate for small inaccuracies of the $\hat{Q}_1$ model.

Algorithm 8: Deep Q-Learning Selection with double Q-function

```
def select($G, \Pi_k$):
    $u_1 \leftarrow \operatorname{argmax}_v \hat{Q}_1((G, \Pi_k), v)$
    $u_2 \leftarrow \operatorname{argmax}_v \hat{Q}_2((G, \Pi_k, u_1), v)$
    $\Pi'_k \leftarrow \text{swap}(\Pi_k, u_1, u_2)$
    $u'_1 \leftarrow \operatorname{argmax}_v \hat{Q}_1((G, \Pi'_k), v)$
    $\text{pred}_{\text{next}} \leftarrow \max_v \hat{Q}_2((G, \Pi'_k, u'_1), v)$
    return $u_1, u_2, \text{pred}_{\text{next}}$
```

done in $\mathcal{O}(N)$, if highly optimized (even for GraphNormGCN).

Given an initial state (G, Π_k) and a swap of nodes u_1, u_2 with associated reward r and next state (G, Π'_k), the corresponding target values t_1 and t_2 obtained through the Bellman Equation for the predictions of $\hat{Q}_1$ and $\hat{Q}_2$, respectively, are

$$\begin{aligned} t^{\hat{Q}_1} &= 0 + \lambda \cdot \max_v \hat{Q}_2((G, \Pi_k, u_1), v) \\ t^{\hat{Q}_2} &= r + \lambda \cdot \max_v \hat{Q}_1((G, \Pi'_k), v) \end{aligned}$$

when using discount factor λ and single step rewards. When training, we avoid the extra zero reward step. We also use $\hat{Q}_2$ as an estimation of the future reward, as we expect it to be more accurate because it is based on more information. This yields the following targets we use instead:

$$t^{\hat{Q}_1} = t^{\hat{Q}_2} = t = r + \lambda \cdot \max_v \hat{Q}_2((G, \Pi'_k, u'_1), v)$$

where $u'_1 = \operatorname*{argmax}_v \hat{Q}_1((G, \Pi'_k), v)$. As loss for single swap, we use the sum of the mean squared errors[4] between the two predicted Q-values and the target t:

$$\mathcal{L} = \left(\hat{Q}_1((G, \Pi_k), u_1) - t\right)^2 + \left(\hat{Q}_2((G, \Pi_k, u_1), u_2) - t\right)^2$$

We train as described in Section 5.3.4 and adapt Algorithm 7 accordingly.

Exploration As node selection is now performed in two steps, we differentiate between three types of random exploration: choosing the first node at random, choosing the second node at random and choosing both nodes at random. To do so, we decide with probability $1 - \sqrt{1 - \varepsilon}$ for both nodes independently whether to make a random choice. This yields some random choice with probability ε each swap. By enabling random exploration for only one node, we reduce correlations between the observations of the different Q-models. This likely allows both models to train more effectively. However, when making a random choice for the second node, the resulting experience cannot be used for training $\hat{Q}_1$, as in our calculation of the target values, we implicitly assume that an "optimal" u_2 was chosen. Similarly, we cannot always add the reverse sample (u_2, u_1) to the memory, as done in Algorithm 7: When any node is chosen randomly, u_1 is not necessarily the "optimal" swap partner for u_2 anymore, which violates the same assumption.

[4]mean in the batched case

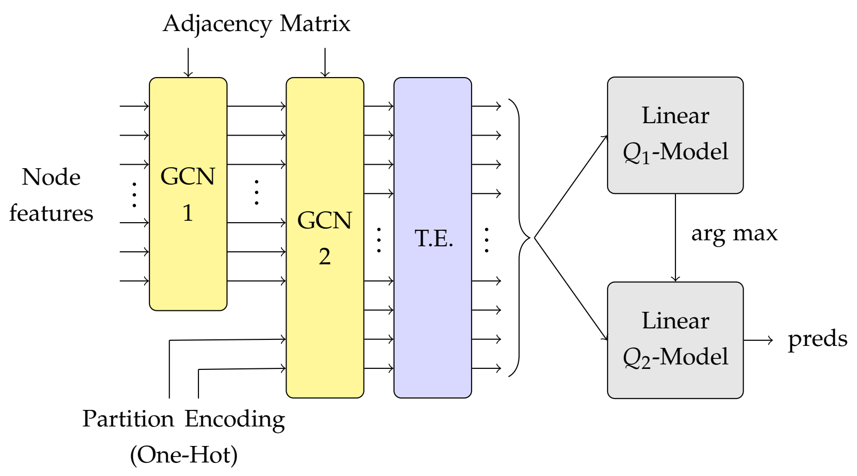


Figure 5.3: The double Q-model architecture for the iterative improvement approach. T.E. denotes Transformer Encoder. The one-hot representation of each node's partition is concatenated with its embedding as input for GCN2.

5.3.7 Diff Values as Features

The node swapping meta algorithm provides an algorithmic framework and ensures perfect balancing. Still, the solution is constructed almost solely based on decisions and computations of the machine learning model with no algorithmic support. This might allow the model to operate very freely, potentially learning entirely new heuristics. However, in the combinatorial setting, some information is much easier computed algorithmically than learned by a neural network. It therefore makes sense to compute such information algorithmically and feed it to the model as input features.

On this account, we extend the inputs to GCN2 by additionally providing the diff values of each node and partition. A diff value $D_v^{V_B}$ is defined as the decrease in cut weight achieved by assigning node v to partition V_B. These values are used in the K/L algorithm and our greedy baseline (see Section 2.3). In fact, the greedy baseline algorithm produces useful partitionings by simply performing greedy decisions based on diff values, which goes to show their power as features. Although the model is already expressive enough to directly compute the number of neighbors a node has in each partition, it cannot compute diff values as easily (see Appendix A.5). We hence expect the extra features to facilitate learning and to increase the model's expressive power.

For $V_A, V_B \in \Pi_k$ and $v \in V_A$ we compute

$$D_v^{V_B} = \sum_{u \in V_B \cap \mathcal{N}(v)} w(\{u, v\}) - \sum_{u \in V_A \cap \mathcal{N}(v)} w(\{u, v\}).$$

Again, we normalize by $\frac{M}{N}$. The values are then concatenated with the node embeddings and the partitioning information and provided to GCN2 as input features. We refer to this extension as DiffFeatures.

Chapter 6

Results

6.1 End-to-end Approach

We train the end-to-end partitioning model on a dataset of graphs with 100 nodes that are generated randomly using all of the four graph generation models described in Section 4.1 and report the loss curves in Figure 6.1. For further details about the dataset, training, and model parameters, see Appendix A.2.

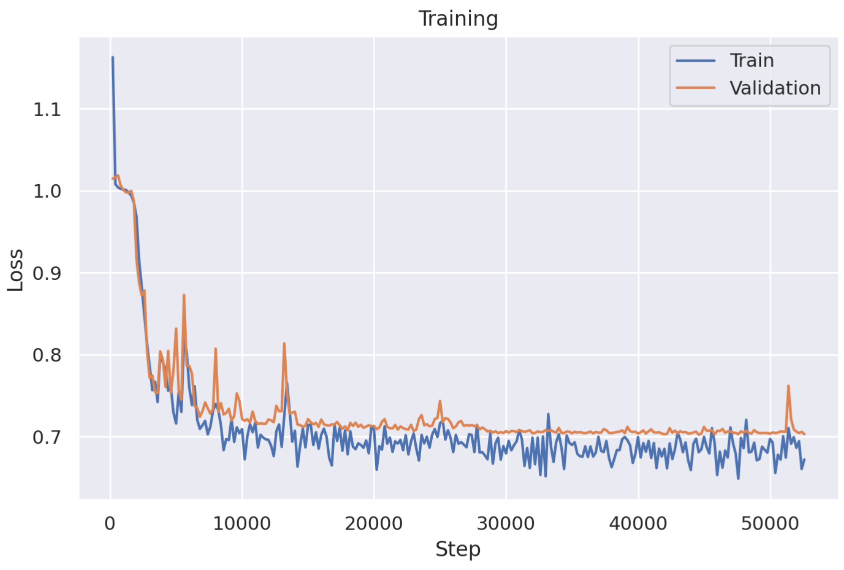


Figure 6.1: Loss for bi-partitioning.

Unless otherwise noted, we report results for bi-partitioning using the model with Transformer Encoder, DegreeSorted features, and a three layer GraphNormGCN. We observe that this model performs consistently well. It significantly outperforms the random baseline on all graphs and generates partitionings that are highly balanced and only slightly heavier than the partitionings generated by the algorithmic baselines (see Table 6.1).

Imbalance	Cut weight: factor to			
	Random	Greedy	K/L	hMETIS
0.0188	0.682	1.097	1.190	1.228

Table 6.1: End-to-end partitioning results on the held-out test set averaged across all graphs.

Density Figure 6.2 shows that the model successfully generates partitionings for graphs of various densities[1]. However, the sparser the graphs, the bigger the difference to the baseline partitioners. In sparser graphs, decisions about single nodes or edges can have a higher influence on the weight of the partitioning. Apparently, this leads to higher variance in results and bigger differences in solution quality. We also suspect that the high importance of single graph features generally makes it harder for an end-to-end machine learning model to generate a good partitioning, as in this setting it is important to make very accurate decisions instead of approximations.

Size Despite the model being trained on graphs of size 100 exclusively, it generalizes well to graphs of various sizes, from 10 nodes up to 2000 nodes. Figure 6.3 shows that the cut weights follow similar trends as the cut weights of the hMETIS partitioner. Especially for the bigger Watts-Strogatz graphs, the model generates partitionings that are only minimally heavier than the partitionings generated by hMETIS. For the very small graphs, the algorithmic solver performs much better as is to be expected. All partitionings are highly balanced, especially on the bigger graphs, except from Barabási–Albert graphs where the imbalance appears to increase as the graphs get very big. Barabási–Albert graphs generally seem more challenging; we observe that, with increasing size, the relative difference to the performance of hMETIS increases and the model performance approaches the random baseline.

[1]Graph density is defined as the ratio of edges to possible edges, i.e. $\frac{2*M}{N(N-1)}$

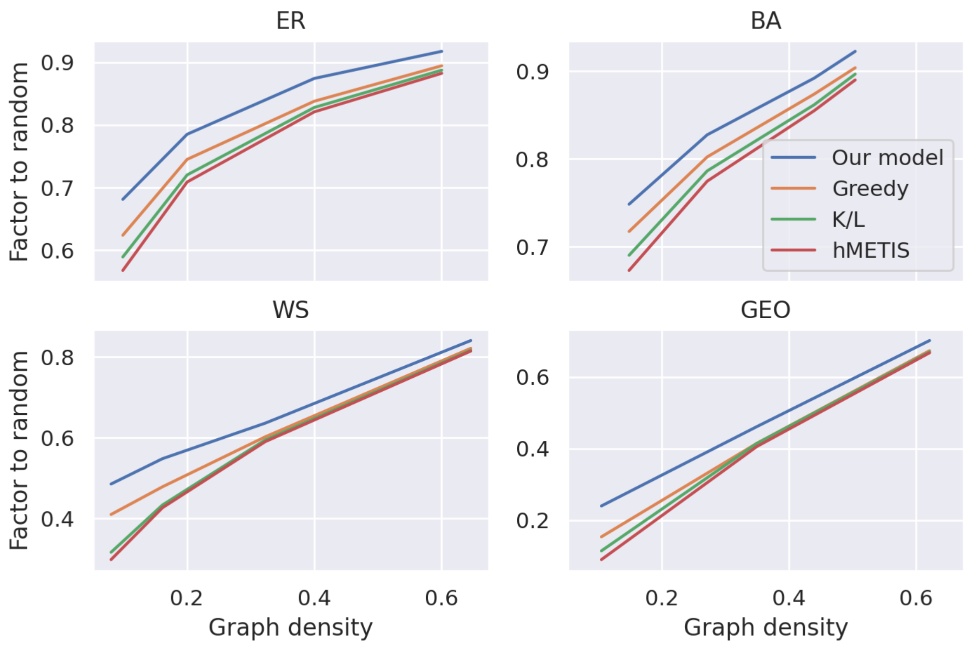


Figure 6.2: Bi-partitioning cut weight by density and graph type in comparison to baselines. (Lower is better)

6.1.1 Node Embeddings

Architecture We compare the three GCN model architectures mentioned in Section 5.1.2 with respect to their effectiveness in generating node embeddings. As described in Section 5.1.2, both MeanGCN (the GraphSAGE style model architecture with mean aggregation) and SymGCN (the symmetrically normalized model architecture) generate node embeddings that lie very close together. We observe that the end-to-end model fails to generate useful partitionings when relying on these architectures (see Table 6.2). The feature-wise variance of the node embeddings across single graphs is very low, which suggests that the node embedding component fails to extract any useful information from the synthetic node features and the graph structure. The resulting partitions are highly imbalanced as the model makes the same prediction for every node.

The GraphNormGCN architecture succesfully adresses this problem by normalizing embeddings across the entire graph. This enables it to produce balanced partitionings based on useful node embeddings. Even if the node embeddings are trained on the partitioning objective, they contain general structural information that can be useful for other tasks: Using a simple XGBoost classifier [8], we are able to achieve an F1 score of 81% for edge prediction on a dataset evenly sampled from the graphs used for testing.

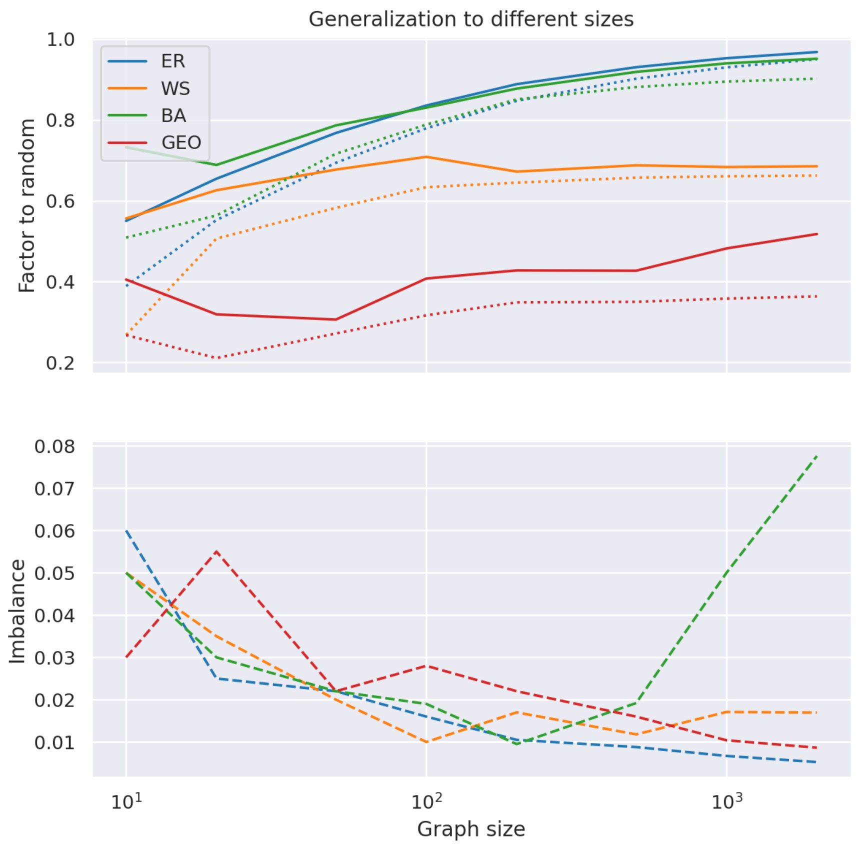


Figure 6.3: Partitioning cut weight (bi-partitioning) and imbalance by graph size and type. The solid lines show the weight of partitionings generated by the model compared to random partitionings. The dotted lines compare hMETIS to random. The model was trained on graphs with 100 nodes. We report the mean of results of runs on 20 graphs per size and type with density ≈ 0.3.

	Imbalance	Embedding std	Factor to hMETIS
MeanGCN	1	0.00067	0.0
SymGCN	1	0.0073	0.0
GraphNormGCN	0.019	0.787	1.228
GraphNormGCN+Pool	0.022	0.785	1.325
GraphNormGCN+Pool+Sum	0.017	0.791	1.225

Table 6.2: Comparison of embedding model architectures. Factor to hMETIS refers to comparison of cut weight. (The GraphNormGCN std is smaller than one, as it is computed after activation function.)

Node Features	Imbalance	Factor to	
		Greedy	hMETIS
One	0.0242	1.178	1.329
Degree	0.0193	1.106	1.240
DegreeSorted	**0.0188**	**1.097**	**1.228**

Table 6.3: Comparison of synthetic node features.

The variation with max pooling for aggregating features of neighboring nodes produces noticeably worse partitionings. Even when applied in addition to aggregation by summation, max pooling only yields marginal performance benefits.

Features When comparing the three different methods for generating synthetic node features, we observe that the end-to-end partitioning model is able to generate useful partitionings based on all three methods (see Table 6.3). However, as expected, the node features that contain more information lead to better partitionings. The novel DegreeSorted features prove particularly useful which suggests that they might also be effective for other problems or model architectures.

6.1.2 Global Information

As described in Section 5.2.2, a GCN extracts local structural information but often cannot provide the global context. The two proposed global components successfully address this problem. As can be seen in Table 6.4, both the GraphEmbedder and the Transformer component increase the model performance notably. Presumably because it is more expressive, the Transformer Encoder enables the model to find slightly better solutions.

Global Component	Imbalance	Factor to	
		Greedy	hMETIS
None	0.041	1.117	1.251
GraphEmbedder	0.029	1.104	1.237
Transformer	**0.019**	**1.096**	**1.228**

Table 6.4: Global components.

k	Factor to		Imbalance	
	Random	hMETIS	Ours	hMETIS
2	0.682	1.23	0.019	0.014
3	0.575	1.02	0.723	0.036
4	0.797	1.37	0.564	0.058
8	0.960	1.49	0.816	0.064

Table 6.5: Mean cut weight and imbalance for bigger k.

6.1.3 Multiple Partitions

Despite generating very balanced useful partitionings when applied for $k = 2$, the model fails to satisfy any reasonable balancing constraint for bigger k (see Table 6.5). Typical constraints for applications require $\text{imb}(\Pi_k) \leq \varepsilon$ where ε is usually a single digit percentage. For bigger k, this is increasingly hard and the end-to-end model fails to generate partitionings that satisfy $\text{imb}(\Pi_k) \leq 0.5$. Interestingly, we still observe a very low balancing loss during training and validation. The model apparently generates predictions with low confidence, which keeps the expected cut balanced. Still, the final cut can be unbalanced as it is generated by assigning each node independently to its most likely predicted partition. Future work could investigate the use of more sophisticated algorithmic methods for generating a partitioning from the model predictions. After all, however, due to the continuitiy of the loss function, the end-to-end model is optimizing a relaxation of the combinatorial problem. This inherent limitation leads to such issues. We aim to overcome this with the Iterative Improvement RL approach.

6.2 Iterative Improvement Approach

Again, we train on a dataset of graphs with 100 nodes that are generated randomly using all of the four graph generation models described in Section 3.1. Based on a random search in the hyperparameter space, we find that the method performs reliably well when using the double Q-functions, 3-step rewards and a discount factor of 0.9. Therefore, we use this configuration as the default, unless otherwise noted. We do not include the DiffFeatures for this configuration (we report results with DiffFeatures in Section 6.2.4). For more information on the dataset, training, and model parameters, see Appendix A.3.

We observe a stable training process as displayed in Figure 6.4. After an immediate significant improvement, the partitioning quality continues to increase slowly but consistently. As the generated partitionings get better, we observe a sudden increase in Q-values which causes the loss to increase in magnitude.

The target Q-values are very close to the more accurate target values we compute in hindsight based on all observed rewards over the course of a run, henceforth hindsight Q-values or hindsight truth (see Appendix A.6). Only after the sudden improvement in the beginning, the target Q-values are slightly overestimated. This does not seem to have a negative effect on overall solution quality.

The generated partitions are always perfectly balanced, as the initial partitioning is balanced and partition size cannot change. The resulting cut weight is consistently low. The results are competitive with the algorithmic greedy approach and close in quality to the K/L algorithm (see Table 6.6). Even if the partitionings generated by hMETIS are lighter, they are not perfectly balanced which makes the comparison slightly unfair. Compared to the end-to-end approach, the iterative improvement approach performs consistently better.

Cut weight: mean factor to				
Random	Greedy	K/L	hMETIS	E2E-Approach
0.690	1.024	1.071	1.104	0.945

Table 6.6: Results of the default configuration on the held out test set.

Density The algorithm performs well for graphs of all four types with varying density (see Figure 6.5). It performs particularly well on sparser graphs, producing better partitionings than the greedy algorithm on sparse Barabási-Albert graphs (where density ≈ 0.15).

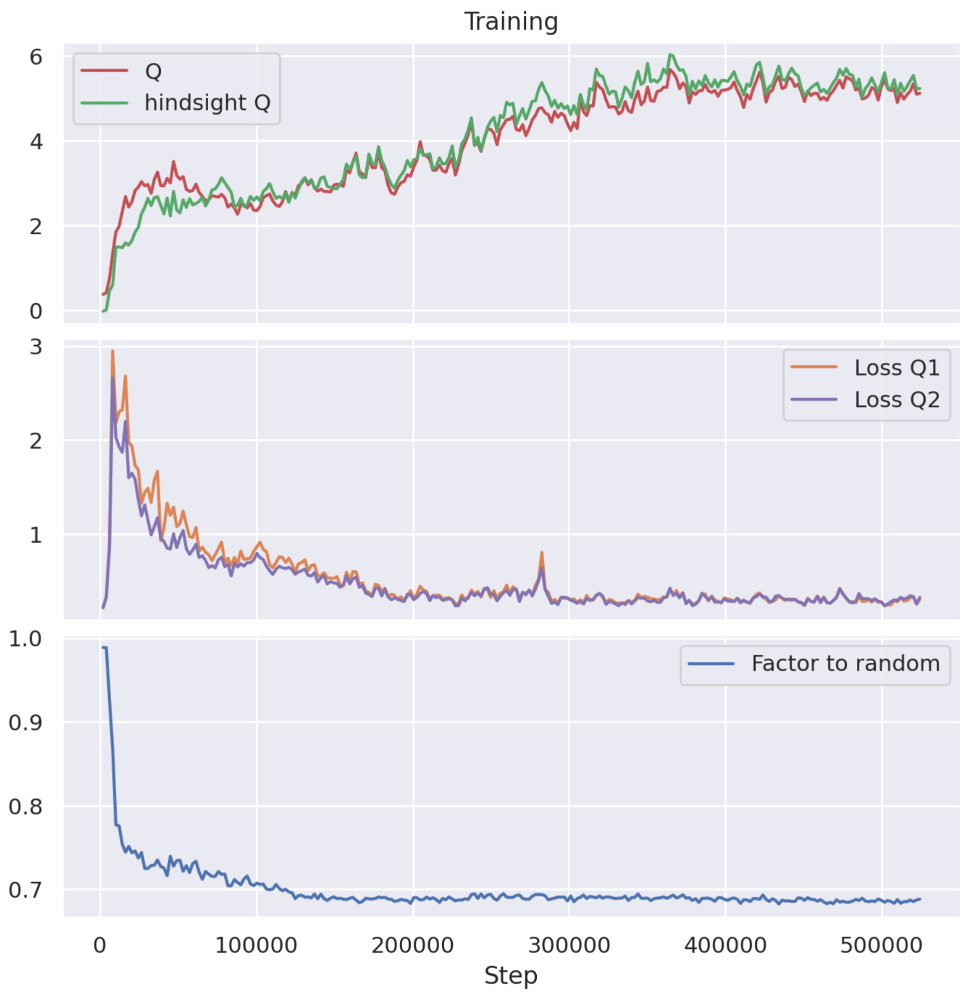


Figure 6.4: Training of the iterative improvement default configuration. All metrics reported on validation set. Factor to random denotes cut weight of the algorithm in comparison to cut weight of random bi-partitionings. Q denotes the target Q-values.

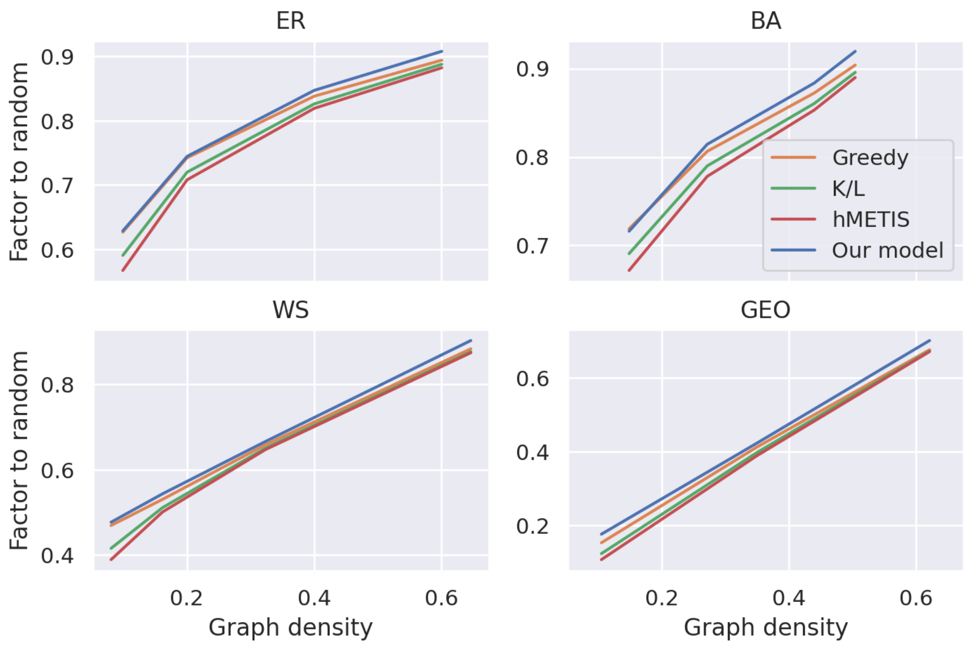


Figure 6.5: Bi-partitioning with default configuration. Cut weight by density and graph type (lower is better). All metrics reported on the held-out test set.

Size The model does not generalize well across graphs of different sizes (see Figure 6.6). Being trained on graphs with 100 nodes, its performance decreases with increasing size difference to the graphs in the training dataset. For graphs with 2000 nodes, the generated partitionings are not better than random partitionings.

6.2.1 Training Subtleties

Overestimated Q-Values As described in Section 5.3.5, we observe overestimated Q-values when not providing the remaining number of algorithm steps as input to the model. Figure 6.7 shows that the Q-values are high until the very end when no information about the algorithm progress is used. This Q-value behavior is inaccurate as it leads to high Q-values even if the algorithm cannot find swaps that further improve the cut weight. It is also undesirable because the algorithm will never terminate early due to negative Q-values, even when the partitioning cannot be further improved. This problem is solved by modeling the underlying MDP correctly to include the step limit and by providing progress information as input to the model: While, without progress information, the algorithm runs until the step limit of 0.5 steps per node despite not improving the partitioning in

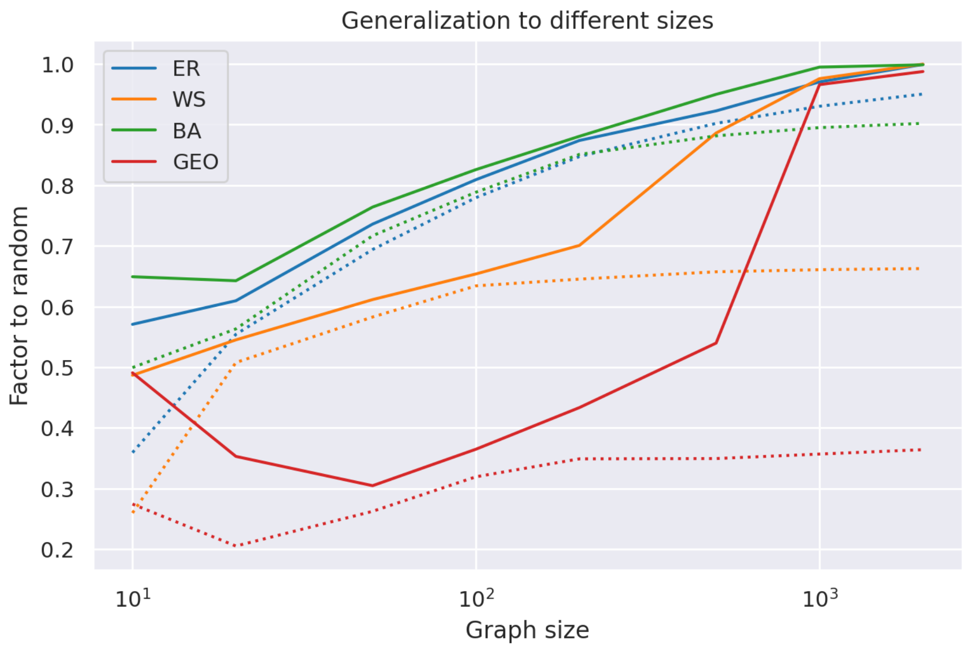
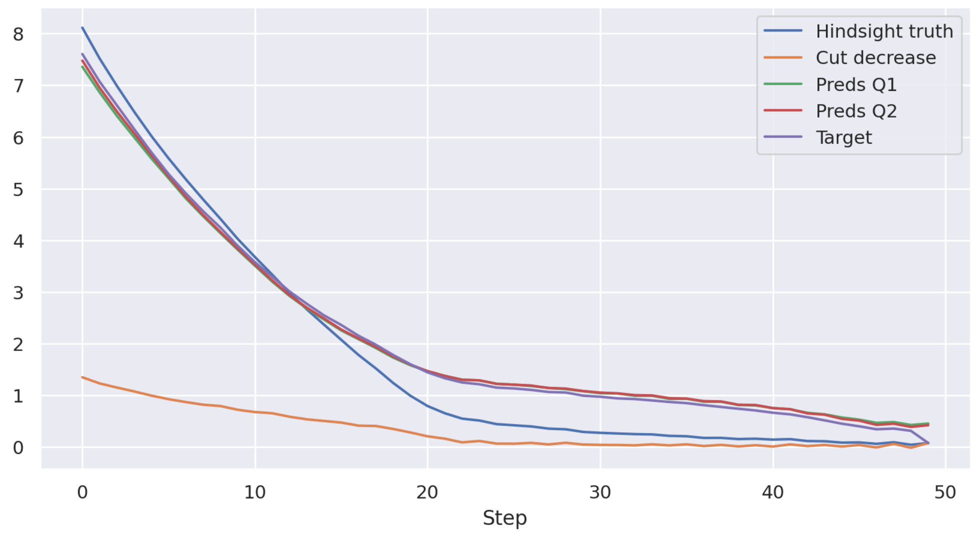


Figure 6.6: Bi-partitioning with default configuration. Results by graph size and graph type. The solid lines show the weight of partitionings generated by the model compared to random partitionings. The dotted lines compare hMETIS to random. The model was trained on graphs with 100 nodes. We report the mean of results of runs on 20 graphs per size and type with density ≈ 0.3.

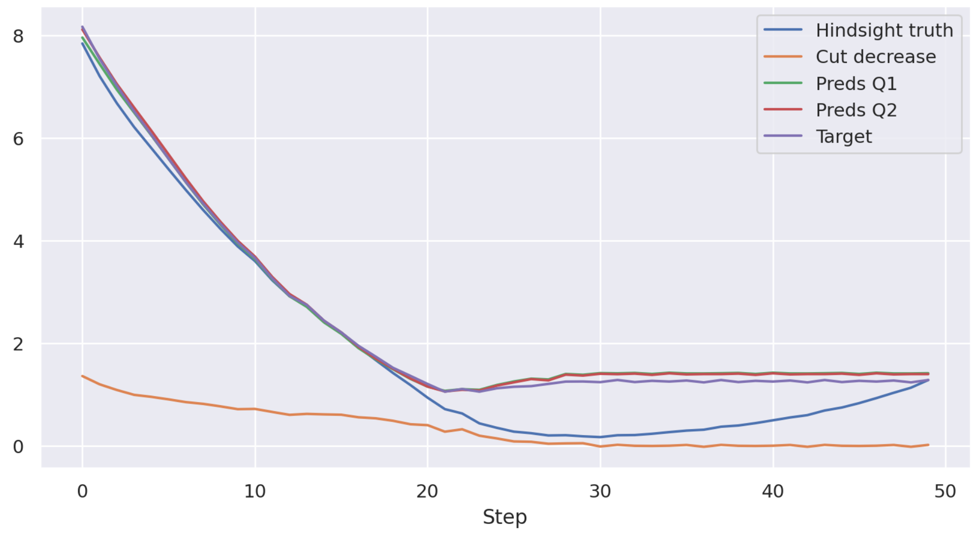
the later steps, it terminates after 0.24 steps per node on average when the progress is provided as input.

Still, we also observe overestimated Q-values when training with single step rewards: The validation set target values based on the one step Bellman equation are around twice as high as the more accurate hindsight Q-values (see Appendix A.6) during most of training. We do not observe this problem when training with n-step rewards for $n \geq 3$: when looking further into the future, the Q-values become more accurate.

Greedyness When training with discount factor 0 and single step rewards, the model is effectively learning the greedy algorithm, i.e. it is learning to directly predict the difference in cut weight resulting from each swap. We observe that it can learn to predict this very accurately. The resulting partitionings are almost on par with the partitionings generated by the greedy algorithm and the predictions are very accurate with a low loss (see Table 6.7).



(a) With progress as input



(b) Without progress as input

Figure 6.7: Mean Q-values on the test set over the course of an algorithm run. Single step rewards.

In order to be able to perform better than a greedy algorithm, the partitioner has to be able to look into the future and plan ahead for more than a single step. This can be achieved by taking into account future rewards during training, either by increasing the discount factor or by using n-step rewards with $n > 1$. The bigger the discount factor, the bigger the incentive to follow a long-term optimal strategy and the less greedy the algorithm. Similarly, bigger n leads to more accurate estimations of future rewards and better long-term results. Consequently, we would like to train with as big of a discount factor and n as possible in order to find the optimal partitionings that can be achieved in the long run.

However, the problem of finding the optimal swap gets significantly harder with an increasing planning horizon. We therefore observe that increasing n or λ does not necessarily lead to performance benefits (see 6.7). Increasing the number of reward steps or the discount factor only brings benefits up to a certain point. In fact, we observe the best performance for single step rewards. For $\lambda = 0.99$ or $n = 10$, the generated partitionings are worse than for the greedier configurations, and we observe higher loss values which can be explained by the increased problem difficulty.

λ	n	Mean Q	Loss	Factor to greedy
0	1	0.664	0.020	1.024
0	3	1.327	0.192	1.005
0.5	3	2.559	0.306	1.013
0.7	3	3.365	0.373	1.019
0.9	1	2.160	**0.186**	**0.994**
0.9	3	5.294	0.789	1.024
0.9	5	4.406	0.675	0.997
0.9	10	5.510	3.018	1.060
0.99	3	5.942	1.483	1.034

Table 6.7: Results for different discount factors λ and reward steps n. All metrics reported on the the held-out test set.

Model Size It might be the case that the model is not expressive enough to make accurate predictions about multiple future steps as it does not have enough trainable parameters. Our model with the standard configuration is relatively small with only 52,530 parameters. However, we experience problems when scaling up the model to more parameters. Bigger models quickly suffer from vanishing gradients, and we are not able to train them with this method and the same hyperparameters. The Pre-LN Transformer only slightly alleviates this issue.

6.2.2 Double Q-Function

For all experiments we use the two step selection method described in Section 5.3.6: We use two Q-functions to select a swap in $\mathcal{O}(N)$ instead of $\mathcal{O}(N^2)$. This enables a roughly five times speedup for training on graphs of size 100. We observe no negative impact on the solution quality. When training with a single Q-function, the model generates partitionings that are 0.693 times lighter than random partitionings, as opposed to 0.690 when using the double Q-function approach. During the training process, we observe no notable differences.

6.2.3 Multiple Partitions

Even for $k > 2$, the iterative improvement approach generates useful partitionings. As shown in Table 6.8, it clearly outperforms the random baseline by a significant margin for $k = 3$ and $k = 4$. However, it cannot compete with the algorithmic hMETIS partitioner in terms of partitioning cut weight. For $k = 8$, it seems that the model fails to learn a useful selection policy, presumably because the problem gets significantly harder for bigger k. Note, however, that our method always produces perfectly balanced partitionings while the hMETIS partitioner produces increasingly imbalanced partitionings, even with the strongest available balancing setting.

k	Factor to		Imbalance	
	Random	hMETIS	Ours	hMETIS
2	0.690	1.104	0	0.016
3	0.751	1.135	0	0.039
4	0.847	1.298	0	0.064
8	0.997	1.371	0	0.068

Table 6.8: Mean cut weight and imbalance for bigger k.

6.2.4 Diff Values

When extending the model input to include DiffFeatures we obtain significantly better results (see Table 6.9). The approach outperforms the greedy baseline, generating paritionings that are 0.978 times lighter on average. Compared to our other methods, the partitionings are also much closer in quality to the partitionings generated by K/L and hMETIS, although still slightly heavier.

As the greedy algorithmic baseline is also directly based on diff values, the iterative improvement approach with DiffFeatures is very similar to it. The decisions are based on the same inputs, however, they are not made greedily but by the machine learning model. It is thus expected that the learned approach performs at least equally good, which it does. By the same principle, providing input features modeled after e.g. K/L or hMETIS might allow a learned approach to outperform even these more advanced algorithms. Such an extension could be an interesting topic of investigation for future work.

	Factor to			
	Random	Greedy	K/L	hMETIS
default	0.690	1.024	1.071	1.104
with DiffFeatures	**0.671**	**0.978**	**1.021**	**1.051**

Table 6.9: Results with DiffFeatures.

Chapter 7

Conclusion

In this work, we propose two methods for employing machine learning for balanced graph partitioning on general, featureless graphs. In the end-to-end method, a machine learning model directly generates partitionings, whereas the iterative improvement method relies on a meta algorithm that makes decisions using a learned heuristic.

We also propose GraphNormGCN, a new GCN architecture that is utilized in both methods. By normalizing across all nodes in the graph instead of performing normalization on a single node basis, the architecture adapts the variance across output vectors to the global graph context and thereby avoids indistinguishable node embeddings. This enables it to extract meaningful structural information from synthetic node features much better than conventional GCN architectures.

The end-to-end model consists of such a GraphNormGCN, followed by a global component like a Transformer Encoder or our proposed GraphEmbedder component, and a linear component that assigns each node to a partition. We train it in a classification-like setup. The model is able to generate balanced partitionings that are significantly lighter than random partitionings across graphs of varying density and size. The partitioning weight is also consistently within few percentage points of the solutions generated by the algorithmic baselines, although never lighter than greedily generated partitionings. For $k > 2$ the method fails to generate useful partitionings. After all, the method is optimizing a continuous relaxation of the combinatorial problem, which likely limits its potential.

We therefore explore the more algorithmically oriented iterative improvement approach. There, an initial balanced partitioning is improved iteratively by swapping nodes' partition assignments. A neural network receives the node features and current partition assignment as input and predicts the value of each swap. Based on this heuristic, the next swap is selected via a greedy policy. By splitting up the Q-function in two parts, we achieve a lower potential asymptotic complexity. We train the model via Deep Q-Learning. The generated partitionings are always perfectly balanced and competitive with the greedy baseline. When providing additional algorithmically computed feature information, we manage to improve upon the greedy baseline and produce partitionings that are close in quality to the partitionings generated by K/L, although never better.

Still, we believe that these approaches are promising. We see three main ways to improve upon our results:

Increased model size The models we use in this work only have around 50,000 parameters. Especially for the iterative improvement approach, we were unable to train bigger models using the same training process. However, bigger, more expressive models might be able to learn better heuristics. In particular, when training with an increased planning horizon, we observe limitations due to small model size. These could potentially be overcome with bigger models.

Stronger algorithmic components Another option is to include stronger algorithmic components. Especially in the combinatorial setting, many computations are hard to learn but can be performed algorithmically. In order to achieve optimal results, as many of these computations as possible should be performed algorithmically to remove a burden from the machine learning components and to make optimal use of their heuristics.

For example, when generating a partitioning from the predictions of the end-to-end approach, an algorithm that ensures balancedness and performs a final optimization step might bring benefits over simply independently assigning each node to its most likely partition, which is done currently. Another interesting idea that accounts for dependencies between different nodes is to assign only the most confidently predicted nodes and to run another iteration with these nodes' partition assignment being already fixed.

For the iterative improvement approach, we directly observe the benefits of stronger algorithmic support. The default method where the model predictions are entirely based on learned features does not consistently outperform the greedy baseline. However, when essentially building upon the greedy algorithm as a basis by providing diff values as input, we observe a notable improvement over the greedy baseline. Similarly, applying machine learning methods by building upon stronger algorithms like hMETIS might even yield improvements over these state of the art methods.

Task specific models In this work, we concentrate on the general case of featureless graphs that are not bound to a certain application. However, the specialized case where graphs of a particular application have to be partitioned is also highly interesting. These graphs often follow a similar distribution. Furthermore, the underlying data often naturally defines node features. These features can allow insights into the corresponding node's role in the graph and its neighborhood, even without explicit investigation of the associated graph structure. Machine learning models are ideally suited to pick up on such information and the underlying distributions, whereas algorithmic solvers cannot utilize such information. The application-oriented case is therefore a very natural application for machine learning methods.

All in all, we see great potential for the application of machine learning to balanced graph partitioning and hard combinatorial optimization problems in general. We hope that future work can build upon our methods to soon achieve significant advancements in these fields.

Appendix A

Appendix

A.1 GCN architectures

In the following we describe the propagation rules of the two common GCN architectures mentioned in 5.1.2. Note that in practice these are implemented much more efficiently with parallelized matrix operations.

Algorithm 9: MeanGCN

```
$\text{agg}_u^{(l)} \leftarrow \frac{1}{\deg(u)} \sum_{v \in \mathcal{N}(u)} h_v^{(l)}$
$x_u^{(l)} \leftarrow \text{elu}\left(W^{(l)} \text{agg}_u^{(l)}\right)$
$h_u^{(l+1)} \leftarrow x_u^{(l)} / \|x_u^{(l)}\|$
```

Algorithm 10: SymGCN

```
$\text{agg}_u^{(l)} \leftarrow \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{1}{\sqrt{\deg(v) \cdot \deg(u)}} \cdot h_v^{(l)}$
$h_u^{(l+1)} \leftarrow \text{elu}\left(W^{(l)} \text{agg}_u^{(l)}\right)$
```

A.2 Parameters End-to-End Approach

A.2.1 Data

The dataset consists of 2700 random graphs from all four graph generation models with 27 different hyperparameter combinations (see Table A.1). We split the data into train, validation, and test set of relative size 0.7, 0.1, and 0.2, respectively.

Generation Model	Parameters
ER	$p \in \{0.1, 0.2, 0.4, 0.6\}$
BA	$m \in \{8, 16, 32, 48\}$
WS	$m \in \{8, 16, 32, 64\}, p \in \{0.1, 0.2, 0.4, 0.8\}$
GEO	$r \in \{0.2, 0.4, 0.6\}$

Table A.1: Parameters for random graph generation. All graphs have 100 nodes.

When analyzing generalization ability to different sizes in Figure 6.3, we evaluate on graphs with density ≈ 0.3 and sizes from 10 to up to 2000 nodes (see Table A.2).

Generation Model	Parameters
ER	$n \in \{10, 20, 50, 100, 200, 500, 1000, 2000\}$, $p = 0.3$
BA	$(n, m) \in \{(10, 2), (20, 3), (50, 9), (100, 18), (200, 37), (500, 92), (1000, 184), (2000, 367)\}$
WS	$(n, m) \in \{(10, 3), (20, 6), (50, 15), (100, 30), (200, 60), (500, 150), (1000, 300), (2000, 600)\}$, $\beta = 0.4$
GEO	$n \in \{10, 20, 50, 100, 200, 500, 1000, 2000\}$, $r = 0.36$

Table A.2: Parameters for random graphs of different sizes.

A.2.2 Model

The model consists of a three-layer GraphNormGCN with hidden layers of dimension 16 and 32 and 64-dimensional output. Based on these embeddings, the final predictions are generated by two linear layers with hidden dimension of size 64, ReLU activation, and no normalization component.

For the Transformer component we use a two-layer PreLN Transformer with hidden layer size 256.

The GraphEmbedder component consists of three parts: The four queries are generated by three linear layers with dimension 256, ReLU activation and layer normalization. A multi-head scaled dot product attention mechanism then generates an output vector for each query. These vectors are concatenated and combined into the final 128-dimensional graph embedding by a single linear layer with layer normalization.

For regularization, we use a dropout rate of 0.1 in the Transformer and 0.2 in all other layers.

A.2.3 Training

We train for 100 epochs with early stopping after $32,000$ steps and report results on the test set at the step with minimum validation loss. We use the Adam optimizer [32] with weight decay of $1\mathrm{e}{-5}$ and a learning rate of 0.001 that is decayed by half every 15 epochs. We use a batch size of 8 graphs, i.e. 800 nodes are processed in parallel.

A.3 Parameters Iterative Improvement Approach

A.3.1 Data

The dataset consists of 1500 [1] random graphs from all four graph generation models with 15 different parameter combinations (see Table A.3). We split the data into train, validation, and test set of relative size 0.7, 0.1, and 0.2, respectively. When analyzing generalization ability to different sizes in Figure 6.6, we evaluate on the graphs in Table A.2.

Generation Model	Parameters
ER	$p \in \{0.1, 0.2, 0.4, 0.6\}$
BA	$m \in \{8, 16, 32, 48\}$
WS	$m \in \{8, 16, 32, 64\}, p = 0.4$
GEO	$r \in \{0.2, 0.4, 0.6\}$

Table A.3: Parameters for random graph generation. All graphs have 100 nodes.

A.3.2 Training

The model is trained on 32 batches of size 128 every 512 algorithm steps, as soon as the replay memory contains at least 2048 samples. The target network is updated every 2048 algorithm steps, i.e. after 128 batches of training. In other work this is sometimes done less frequently. We opt for a higher update frequency in order to speed up training and to achieve higher sample efficiency, as, in our setting, exploration is time consuming.

The learning rate is set to 0.001 in the beginning and decayed by half every 120,000 algorithm steps. We also use a schedule to decrease the exploration parameter ε over time in order to improve exploitation in later stages of training (see Table A.4).

Algorithm Step (divided by 2048)	0	40	55	70	85	100
ε	0.8	0.6	0.4	0.2	0.1	0.01

Table A.4: Exploration Schedule

[1]Due to slower inference time, we use less different graph types than in the end-to-end approach in order to reduce time spent on validation.

A.3.3 Model

We use a relatively small model with 52,530 parameters. The hidden layers of the three layer GraphNormGCN have dimension 16 and 32 and it produces 32-dimensional node embeddings. Together with the one-hot encoding of the respective partitions, these are fed to the single layer GCN2 that again produces 32-dimensional outputs. The PreLN Transformer Encoder has 4 layers and a hidden size of 64. Both the Q1 and the Q2 model consist of a Tanh activation function followed by two-layer feed-forward network with hidden layer size 128 and ReLU activation. For regularization we use dropout 0.1 in the Transformer and 0.2 in all other layers.

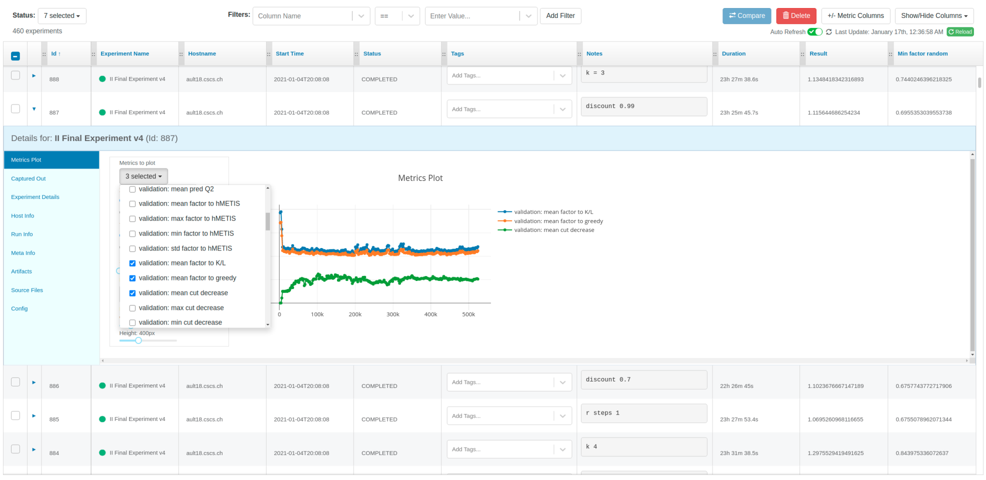
A.3.4 Pretraining

We pretrain the GraphNormGCN as a component of the end-to-end approach, expecting to see faster convergence and better results. However, in practice, we observe no benefits of the pretrained model over a randomly initialized model.

A.4 Implementation and Setup

All models are implemented using PyTorch [46] and HuggingFaces's Transformers library [63]. For the end-to-end approach we train on an NVIDIA V100 accelerator, while the iterative improvement approach does not benefit from GPU due to the associated data copying overhead and due to sequential algorithmic computations. If parallelized and optimized, the training of the iterative improvement approach could likely be sped up significantly.

During training we log up to 71 different metrics, as well results by graph type, visualizations of gradient flow, and visualizations of Q-value development over single algorithm runs. To log these metrics, we use the Python library Sacred [34]. All logs are saved on a MongoDB hosted on a Google Cloud server. This allows us to keep our logs centralized, in an organized way, even when training on different machines. The Omniboard web dashboard allows us to conveniently access these logs (see Figure A.1).



460 experiments

Last Update: January 17th, 12:36:58 AM

Id ↑	Experiment Name	Hostname	Start Time	Status	Tags	Notes	Duration	Result	Min factor random
888	II Final Experiment v4	ault18.cscs.ch	2021-01-04T20:08:08	COMPLETED	Add Tags...	k = 3	23h 27m 38.6s	1.1348418342316893	0.7440246396218325
887	II Final Experiment v4	ault18.cscs.ch	2021-01-04T20:08:08	COMPLETED	Add Tags...	discount 0.99	23h 25m 45.7s	1.115644686254234	0.6955353039553738

Details for: **II Final Experiment v4** (Id: 887)

- [ ] validation: mean pred Q2
- [ ] validation: mean factor to hMETIS
- [ ] validation: max factor to hMETIS
- [ ] validation: min factor to hMETIS
- [ ] validation: std factor to hMETIS
- [x] validation: mean factor to K/L
- [x] validation: mean factor to greedy
- [x] validation: mean cut decrease
- [ ] validation: max cut decrease
- [ ] validation: min cut decrease

Metrics Plot

- validation: mean factor to K/L
- validation: mean factor to greedy
- validation: mean cut decrease

Id ↑	Experiment Name	Hostname	Start Time	Status	Tags	Notes	Duration	Result	Min factor random
886	II Final Experiment v4	ault18.cscs.ch	2021-01-04T20:08:08	COMPLETED	Add Tags...	discount 0.7	22h 26m 45s	1.1023676667147189	0.6757743772717906
885	II Final Experiment v4	ault18.cscs.ch	2021-01-04T20:08:08	COMPLETED	Add Tags...	r steps 1	23h 27m 53.4s	1.0695260968116655	0.6755078962071344
884	II Final Experiment v4	ault18.cscs.ch	2021-01-04T20:08:08	COMPLETED	Add Tags...	k 4	23h 31m 38.5s	1.2975529419491625	0.843975336072637

Figure A.1: Overview over experiments.

A.5 Iterative Improvement Model Expressiveness

The model we use for the iterative improvement approach relies on node embeddings generated by the GraphNormGCN component. As described in Section 6.1.1, these contain useful structural information and can even be used for edge prediction. However, apart from the graph structure, information about the current partitioning is also crucial for predicting Q-values. This information is provided as a one-hot encoding of each node's current partition and input to GCN2[2]. This enables the GCN to e.g. immediately compute the number of neighbors each node has in each partition:

Let h_u be the feature vector input to GCN2 for node u. Then h_u consists of the output of GCN1 of length l concatenated with the one-hot encoding of node u's partition of length k. I.e.

$$h_u[l+i] = \begin{cases} 1 & \text{if } u \text{ in partition } P_i \\ 0 & \text{otherwise} \end{cases}$$

for $1 \leq i \leq k$ assuming 1-based indexing. GCN2 then aggregates

$$s_u \leftarrow \sum_{v \in \mathcal{N}(u)} h_v$$

and we have

$$s_u[l+i] = \sum_{v \in \mathcal{N}(u)} h_v[l+i] = \#\text{neighbors of } u \text{ in } P_i.$$

On the contrary, the diff values that prove to be useful in the DiffFeatures extension cannot be obtained as easily. Being the difference between the neighbors in each partition and the neighbors in the nodes' own partition they could e.g. be obtained via

$$s_u[l:] - M\,(h_u[l:] \odot s_u[l:])$$

where M is the $k \times k$ matrix full of ones and $[l:]$ denotes all elements but the first l. In any case, a multiplication between two vectors derived from the input is required to compute the diff values exactly when node u could be in any partition. The GCN cannot directly perform such operations. However, the Transformer that follows is more expressive, and it is well known that even a linear neural network like the final Q-model component can approximate any function if wide enough [23]. Still, even if the model might be able to approximate them, our results suggest that it is useful to provide the diff values as input.

[2]When using the DiffFeatures extension, this information is also contained in the diff values. Here, we just consider the base architecture without DiffFeatures.

A.6 Hindsight Targets

In the iterative improvement approach, we use n-step Q-learning for training the model. This means that, for a state-action pair (s_i, a_i), the target Q-value t_i is computed via the Bellman equation based on the n next rewards and the maximum predicted Q-value[3] q'_{max} for state s_{i+n}:

$$t_i = \sum_{j=i}^{i+n-1} r_j + \lambda \cdot q'_{\text{max}}$$

Note that q'_{max} serves as an approximation of the future lifetime reward when executing a greedy policy using the Q-model and discounting by λ. This estimation might be inaccurate, so we have to exercise caution when evaluating model performance with respect to the target values computed based on this estimation. However, during validation, when no exploration is performed, we can compute a more accurate estimation of this future reward: In hindsight, after executing such a greedy policy, the future rewards are known. We hence define the hindsight targets (or hindsight truth / hindsight Q-values) by recursively replacing q'_{max} with its target, if available: We compute the hindsight target at step i as

$$t_i^{\text{hindsight}} = \begin{cases} \sum_{j=i}^{i+n-1} r_j + \lambda \cdot t_{i+n}^{\text{hindsight}} & \text{if } i \leq \#\text{steps} - n \\ t_i & \text{otherwise} \end{cases}$$

and use these values to evaluate the accurateness of the target Q-values t during validation.

[3]The precise definition of q'_{max} depends on whether we use the double Q-function approach.

Bibliography

[1] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[2] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[3] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

[4] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

[5] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[6] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[7] Aydin Buluç, Henning Meyerhenke, Ilya Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. *ArXiv*, abs/1311.3144, 2016.

[8] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[9] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711, 2016.

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[11] Shantanu Dutt. New faster kernighan-lin-type graph-partitioning algorithms. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 370–377. IEEE, 1993.

[12] Ulrich Elsner. Graph partitioning-a survey. 1997.

[13] P Erdős and A Rényi. On random graphs i. *Publ. math. debrecen*, 6(290-297):18, 1959.

[14] Charbel Farhat. A simple and efficient automatic fem domain decomposer. *Computers & Structures*, 28(5):579–602, 1988.

[15] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *19th design automation conference*, pages 175–181. IEEE, 1982.

[16] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, page 47–63, New York, NY, USA, 1974. Association for Computing Machinery.

[17] Olivier Goldschmidt and Dorit S. Hochbaum. A polynomial algorithm for the k-cut problem for fixed k. *Mathematics of Operations Research*, 19(1):24–37, 1994.

[18] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[19] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[20] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[21] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

[22] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.

[23] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[24] L[aurent] Hyafil and R[onald] L. Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. Technical Report Rapport de Recherche no. 33, IRIA – Laboratoire de Recherche en Informatique et Automatique.

[25] David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, January 2000.

[26] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[27] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[28] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[29] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[30] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[31] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung-Ro Moon. Genetic approaches for graph partitioning: a survey. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 473–480, 2011.

[32] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[33] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[34] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49 – 56, 2017.

[35] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[36] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[37] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[38] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The case for a learned sorting algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1001–1016, 2020.

[39] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pages 539–548, 2018.

[40] Akash Mittal, Anuj Dhawan, Sahil Manchanda, Sourav Medya, Sayan Ranu, and Ambuj Singh. Learning heuristics over large graphs via deep reinforcement learning. *arXiv preprint arXiv:1903.03332*, 2019.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[42] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

[43] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9839–9849, 2018.

[44] Azade Nazi, Will Hang, Anna Goldie, Sujith Ravi, and Azalia Mirhoseini. Gap: Generalizable approximate graph partitioning framework. *arXiv preprint arXiv:1903.00614*, 2019.

[45] Vol Nr and Per-Olof Fjallstrom. Algorithms for graph partitioning: A survey. 10 1998.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[47] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference on High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[48] Bo Peng, Lei Zhang, and David Zhang. A survey of graph theoretical approaches to image segmentation. *Pattern recognition*, 46(3):1020–1038, 2013.

[49] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[50] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933, pages 164–175. Springer, 2013.

[51] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[52] 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):484–489, Jan 2016.

[53] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan

Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[54] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, July 1997.

[55] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[56] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[57] Mikkel Thorup. Minimum k-way cuts via deterministic greedy tree packing. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, page 159–166, New York, NY, USA, 2008. Association for Computing Machinery.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[59] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in neural information processing systems*, pages 2692–2700, 2015.

[60] Chris Walshaw and Mark Cross. Jostle: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques*, pages 27–58, 2007.

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

[62] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[63] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

[64] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, March 2020.

[65] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 10524–10533. PMLR, 2020.

[66] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings, 2016.

[67] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):11, 2019.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.