Multi-Way Max-Cut Using GCNs

Mohammad Javaad Akhtar, Roberto Solis-Oba

Abstract

Combinatorial optimization (CO) problems are prevalent in many scientific and industrial applications. In the multi-way max-cut problem, the objective is to divide a weighted, undirected graph $G = (V, E)$ with a set $T \subseteq V$ of k terminals into k partitions such that each terminal is in a different partition and the total weight of the edges having endpoints in different partitions is maximum. In the balanced version of the problem, each partition must have equal number of nodes as well.

This work introduces a learning-based framework for solving these two and other clustering NP-hard problems using Graph Convolutional Neural Networks (GCNs). Our approach leverages a relaxation of the problems to define differentiable loss functions, enabling unsupervised training of the GCN. Upon completion of training, a simple projection step and post-processing is applied to obtain integer solutions. Experimental results demonstrate that the method achieves competitive accuracy compared to traditional solvers while offering significant improvements in scalability and speed. Moreover, the approach generalizes effectively across datasets and handles graphs that are orders of magnitude larger than those seen during training. Beyond the multi-way max-cut problem, this approach has the potential to be applied to other network clustering problems, such as community detection, modularity optimization, and graph partitioning in large-scale data networks. The adaptability of the framework suggests promising applications in areas like computational biology, social network analysis, and telecommunications, where efficient graph partitioning is essential.

1 Introduction

Combinatorial optimization (CO) encompasses a broad spectrum of problems, each with unique structures and constraints, yet all share the common goal of identifying optimal solutions within a finite set of alternatives [1]. CO problems are foundational in optimization, arising in diverse application areas such as business analytics, medical diagnostics, supply chain optimization, and engineering design, among others [2]. The critical importance of CO problems has driven extensive research into developing effective algorithms that can efficiently tackle their inherent complexity [10]. Despite their importance, finding solution to NP-hard optimization problem in a reasonable amount of time and with high accuracy remains a challenging task.

As the size of problem instances grows, exhaustive search strategies quickly become computationally infeasible for all but the smallest cases, necessitating the development of more sophisticated approaches capable of producing optimal or near-optimal solutions within reasonable computational limits [10]. Approximation algorithms offer theoretical guarantees for solution quality, but for some problems achieving satisfactory solutions may not even be feasible [3].

Traditional methods for dealing with NP-hard graph optimization problems generally fall into three main categories: exact algorithms, approximation algorithms, and heuristics. Exact algorithms, such as enumeration or branch-and-bound methods, guarantee optimal solutions, but these methods are computationally prohibitive for large problem instances. Approximation algorithms, while computationally efficient, may exhibit weak empirical performance. Heuristics, while fast and versatile, lack theoretical guarantees and often require extensive problem-specific research and experimentation to achieve satisfactory results [5].

Since the seminal work of Hopfield and Tank, the advent of deep learning has reinvigorated interest in neural network approaches for tackling CO problems. Among these, graph neural networks (GNNs) have emerged as a particularly powerful tool due to their ability to learn effective features and representations from graph-structured data [4]. GNNs are specifically designed to handle the unique structure of graphs, enabling them to capture complex relationships

and dependencies between nodes and edges. This success has motivated researchers to explore their application to CO problems involving graphs [1]. Despite this progress, CO problems remain challenging for neural networks due to their discrete nature and the difficulty of bridging the gap between continuous optimization and combinatorial solution spaces [4].

Unsupervised learning frameworks have provided a promising pathway for addressing CO problems, transforming them into optimization problems with differentiable objective functions [2]. This paradigm enables efficient learning on large, unlabeled datasets, but it also introduces challenges, such as the design of suitable loss functions and ensuring convergence to high-quality solutions.

In this paper, we propose an approach to address the challenges of the multi-way max-cut and balanced multi-way max-cut problems using Graph Neural Networks (GNNs). Our contributions are as follows: First, we explore the relationship between the number of nodes in the training graphs and the performance of the neural network, demonstrating that training on small graphs combined with a handful of large graphs yields accurate and scalable results. Second, we study the impact of the degree distribution of nodes on the accuracy of a neural network, showing that the inclusion of a few high-degree graphs in the training set significantly scalability. Third, we propose modifications to the GNN architecture from that of Schuetz et al. [1], including the integration of a customized loss function and an effective post-processing heuristic to enhance partitioning accuracy. We think that these contributions will help advance the applicability of GNNs to combinatorial optimization problems, with a focus on scalability, efficiency, and practical relevance.

2 Related Work

In this section, we review existing research on neural network-based approaches for combinatorial optimization (CO) problems.

2.1 Supervised Learning for CO Problems

Neural network approaches to combinatorial optimization (CO) problems have predominantly relied on supervised learning methods, which aim to approximate complex, non-linear mappings from input representations to solutions by minimizing a handcrafted loss function. Early advancements in this area include Pointer Networks, a neural architecture based on the sequence-to-sequence (Seq2Seq) model, which utilizes an attention mechanism to solve problems like the Travelling Salesman Problem (TSP) [18].

Numerous studies have integrated Graph Neural Networks (GNNs) with various heuristics and search procedures to address CO problems such as quadratic assignment [19], graph matching [20], graph coloring [21], and TSP [23, 22]. By leveraging the structural properties of graphs, GNNs can learn meaningful node and edge representations that aid in optimization. However, as highlighted in [24], the success of supervised learning approaches is contingent on access to large-scale labeled datasets with pre-optimized instances, creating a fundamental challenge: the very need for high-quality training data renders the approach impractical for large NP-hard problems, where computing exact solutions is computationally prohibitive [25].

To address this issue, hybrid approaches have emerged that combine neural networks with traditional solvers. Neurocore [26] enhances SAT solvers by integrating a mixed integer linear program (MILP) for variable branching decisions, while Gasse et al. [28] employ imitation learning with a GNN to approximate branching heuristics in integer programming. Additionally, Wang et al. [29] incorporate an approximate semi-definite programming satisfiability solver as a neural network layer, and Vlastelica et al. [30] propose a differentiable architecture that interpolates the solver's piecewise output. Although these methods bridge the gap between learning-based models and classical optimization techniques, they still require extensive labeled data for effective generalization.

Solving large CO problems optimally can take hours or even days, making dataset generation infeasible for real-world applications [25]. Moreover, even when labeled datasets are available, they are often biased or non-representative, limiting the model's ability to generalize to unseen problem instances [25]. These challenges have motivated research into unsupervised and reinforcement learning as alternative paradigms, which eliminate the need for labeled data but

introduce their own difficulties, such as high variance in training and inefficient sampling.

2.2 Reinforcement Learning for Combinatorial Optimization

Reinforcement learning (RL) has emerged as an alternative approach to combinatorial optimization problems, addressing the need for labeled datasets by learning a policy that optimizes an expected reward function [1]. In this context, the problem's native objective function often serves as the reward, allowing the possibility to iteratively refine decisions [32].

One of the earliest applications of RL in CO extended pointer networks to an *actor-critic* framework, where Bello et al. [31] trained an approximate TSP solver using a recurrent neural network (RNN) encoder and used the expected tour length as a reward function. Subsequent works incorporated graph attention networks to improve accuracy on two-dimensional Euclidean TSP, achieving near-optimal results for graphs up to 100 nodes [33]. Additionally, RL-based approaches have been used to tackle TSP variants with hard constraints by employing a multi-level hierarchical framework, where each layer learns distinct policies to enhance solution quality [34].

More recently, Dai et al. [35] trained a deep Q-network (DQN) to incrementally construct solutions to NP-hard graph problems, demonstrating superior performance compared to previous learning-based techniques. Their approach integrates RL with graph embeddings to learn efficient greedy meta-heuristics, achieving competitive results for Minimum Vertex Cover, MaxCut, and TSP on graphs with up to 1200 nodes. Similarly, Khalil et al. [35] proposed a Q-learning framework combining greedy heuristics with structure2vec embeddings, which showed promising results for max-cut, minimum vertex cover, and TSP. Q-learning has also been employed for the maximum common subgraph problem [36], while policy gradient methods incorporating attention mechanisms have been successfully applied to TSP and vehicle routing [37, 38, 39].

Despite eliminating the dependency on labeled data, RL-based approaches face several well-documented challenges:

- *Sample Inefficiency:* RL requires a substantial number of training episodes to converge, increasing computational costs [40, 41].
- *Training Instability:* Poor gradient estimation, sensitivity to initial conditions, and correlated sequential observations contribute to unstable training [40, 42].
- *Exploration Challenges:* Effective exploration of vast solution spaces remains difficult, particularly for large-scale CO problems [2].

To mitigate these challenges, hybrid approaches have been explored by integrating RL with additional optimization techniques. Yolcu and Poczos [43] employed the REINFORCE algorithm to learn local search heuristics for SAT problems, incorporating curriculum learning to enhance stability. Chen and Tian [44] introduced an actor-critic framework to iteratively refine combinatorial solutions. Additionally, Wang et al. [29] incorporated an approximate SDP satisfiability solver as a neural network layer, blending deep learning with mathematical optimization [17].

While RL provides a promising alternative to supervised learning, its reliance on large state spaces, lack of full differentiability, and high computational costs make it challenging to train effectively [25]. These limitations have spurred growing interest in unsupervised learning methods, which aim to learn optimium structures without the need for explicit labels or reward signals.

2.3 Unsupervised Learning for CO Problems

Unsupervised learning, as utilized in this paper, provides a promising alternative to supervised and reinforcement learning approaches for solving combinatorial optimization (CO) problems. Unlike supervised methods, which require large labeled datasets, unsupervised learning models directly optimize a differentiable objective function whose minima correspond to the discrete solution space of the CO problem [54, 53]. This approach allows training on large, unlabeled datasets, making it particularly suitable for large-scale instances where generating optimal labeled solutions is computationally prohibitive.

One of the primary techniques in unsupervised learning for CO problems is the continuous relaxation of discrete objective functions. By defining a smooth, differentiable surrogate of the original objective functions, optimization can be performed efficiently using gradient-based methods [55, 54]. For instance, previous work has applied unsupervised

learning to train Graph Neural Networks (GNNs) for the max-cut problem by optimizing a relaxation of its objective function [6]. These relaxation-based methods approximate the discrete solution space, enabling efficient training without requiring explicit labels. However, relaxation techniques often introduce challenges, such as degenerate solutions and difficulty in mapping soft assignments back to valid discrete solutions [22]. To mitigate these issues, empirically identified correction terms and auxiliary loss functions are frequently incorporated [53].

Another line of research has explored the use of physics-inspired frameworks in conjunction with unsupervised learning. Models based on Ising systems and quantum annealing provide natural formulations for many CO problems, leveraging inherent physical constraints to guide the learning process [1]. These approaches embed combinatorial problems into differentiable frameworks that generalize across problem instances, allowing for improved solution quality.

Graph Neural Networks for CO Problems

Graph Neural Networks (GNNs) have become a cornerstone in the study of CO problems involving graph structures. Their ability to learn complex node and edge relationships makes them particularly well-suited for problems such as max-cut, graph partitioning, and clustering [4]. Notably, RUN-CSP, a recurrent GNN model, has been successfully applied to constraint satisfaction problems, including Maximum Independent Set, Maximum 2-SAT, 3-colorability, and MaxCut, achieving competitive performance against traditional heuristics and semi-definite programming-based methods [16]. Similarly, Yao et al. trained a GNN to solve the max-cut problem using either policy gradients or smooth relaxations of the cut objective, demonstrating promising results for small graphs with up to 500 nodes [17].

Despite these advancements, unsupervised learning for CO problems remains challenging due to the highly non-convex nature of the objective functions. Without explicit labels, neural networks can become trapped in poor local optima, limiting solution quality [17]. Furthermore, decoding valid discrete solutions from the soft assignments of a neural network remains an open problem, particularly for constrained CO formulations [22].

Our Contribution

In this paper, we extend prior work by incorporating unsupervised learning to solve the *multi*-way max-cut problem. Our approach leverages relaxation-based optimization combined with an efficient post-processing heuristic to recover high-quality discrete solutions. By training on unlabeled datasets and generalizing across different graph structures, we demonstrate that our method achieves competitive accuracy and scalability, even for large graphs. This work builds upon previous research [1] while addressing key challenges in scalability, discrete solution recovery, and computational efficiency for combinatorial optimization problems.

The main contributions of this work include the development of a Graph Neural Network (GNN)-based framework for solving the multi-way and balanced multi-way max-cut problems in an unsupervised learning setting, eliminating the need for labeled training data. We introduce a differentiable loss function that relaxes the combinatorial optimization objective, enabling gradient-based learning while preserving the problem's discrete nature. Additionally, we incorporate an efficient post-processing heuristic that refines the neural network's probabilistic outputs, ensuring high-quality discrete solutions with minimal computational overhead. Our experimental analysis provides insight into the impact of graph properties, such as node degree and training graph size, on the model's generalization capabilities, demonstrating that training on small graphs combined with few larger graphs enables effective scaling to significantly larger graphs. Furthermore, we show that our method achieves a balance between solution accuracy and computational efficiency, significantly outperforming traditional integer problem solvers in terms of speed while maintaining competitive accuracy. Beyond the multi-way max-cut problem, our approach could be applied to other network clustering tasks, such as community detection, modularity optimization, and large-scale graph partitioning, highlighting its potential for real-world combinatorial optimization challenges.

3 Background

To establish the necessary notations and terminology, we provide a brief review of the combinatorial optimization problems studied in this work, specifically the multi-way max-cut and balanced multi-way max-cut problems. Additionally, we introduce Graph Convolutional Networks (GCNs), which serve as the foundation for our proposed learning-based approach. These concepts are essential for understanding the methodology and experimental framework presented in later sections. For further clarification of technical terms, highlighted in *italics*, additional definitions are provided in the glossary section at the end of this paper.

3.1 Multi-Way Max-Cut

The *multi*-way max-cut problem is a generalization of the classic max-cut problem. Given an undirected, weighted graph $G = (V, E)$, where V is the set of vertices and E is the set of edges with weights $w : E \to \mathbb{R}^+$, and a set of terminals $S = \{s_1, s_2, \ldots, s_k\} \subseteq V$, a *multi-way max-cut* is a subset of edges $E' \subseteq E$ such that removing E' from G separates all terminals. The objective of the **multi-way max-cut problem** is to find a subset of edges E' of maximum total weight whose removal ensures that each terminal belongs to a separate partition.

3.1.1 Applications of Multi-Way Max-Cut

The multi-way max-cut problem has numerous applications in various fields, including:

- *VLSI Design:* Used in circuit partitioning to minimize cross-chip communication and optimize performance [50].
- *Network Optimization:* Used in designing fault-tolerant networks by ensuring minimal inter-group connectivity [51].
- *Machine Learning and Clustering:* Utilized for clustering high-dimensional data by ensuring well-separated partitions [48].
- *Statistical Physics:* Used in modeling spin glass problems and phase transitions in complex physical systems [50].
- *Image Processing:* Applied to segmentation tasks by partitioning images into meaningful regions based on edge weights [50].
- *Communication Networks:* Used in the design of robust and efficient communication networks to minimize interference [52].
- *Social Network Analysis:* Provides a standard for network classification and robustness evaluation in large-scale social networks [51].

3.2 Balanced Multi-Way Max-Cut

The objective of the **balanced multi-way max-cut problem** is to find a subset of edges E' of maximum total weight that partitions the vertices into subsets $V_1, V_2, \ldots, V_k$ contains approximately the same number of nodes and such that each terminal belongs to a different partition.

3.2.1 Applications of Balanced Multi-Way Max-Cut

The balanced multi-way max-cut problem has applications in various domains, including:

- *Load Balancing in Distributed Computing:* Ensuring fair distribution of computational tasks across different processors improves efficiency and reduces bottlenecks [45].
- *Network Partitioning:* Used in designing network topologies where minimizing inter-group communication while maintaining balanced subnetworks is crucial [46].
- *Community Detection in Graphs:* Identifies well-separated, equally sized communities in social networks, improving the quality of clustering for large-scale networks [47].
- *Cluster-Based Optimization:* Applied in clustering high-dimensional datasets while ensuring uniform cluster sizes, particularly in machine learning and data analysis [48].
- *VLSI Design:* Helps balance the number of logic gates assigned to different partitions while optimizing circuit communication, which is essential in hardware optimization [49].

3.3 Graph Convolutional Networks (GCNs)

Graph Convolutional Networks (GCNs) are a class of deep learning models specifically designed to process and learn *node representations* from graph-structured data. Unlike traditional deep learning architectures that operate on *Euclidean data* (e.g., images, sequences), GCNs leverage spectral graph theory and local connectivity to propagate information across nodes, making them highly suitable for applications such as node classification, link prediction, community detection, and combinatorial optimization.

A graph $G = (V, E)$ consists of a set of vertices V and edges E, where each node $u \in V$ is associated with a feature vector $x_u \in \mathbb{R}^d$, and each edge $(u, v) \in E$ may also have associated attributes. The goal of GCNs is to iteratively refine *node representations* by aggregating and propagating information from neighboring nodes through spectral or spatial convolutions.

3.3.1 GCN Architecture and Message Passing

A GCN consists of multiple layers, where each layer refines a node's representation by *aggregating information from its local neighborhood* $\mathcal{N}(u)$. At layer l, each node u updates its representation $h_u^{(l)}$ based on the features of its neighbors and its own. This iterative process enables the network to capture hierarchical structural information, akin to how convolutional layers in CNNs expand their receptive field.

A typical GCN layer consists of three main functions [17]:

- **Message Passing:** Each node aggregates feature information from its neighbors through the edges.
- **Aggregation:** The received messages are combined using a normalized sum or mean operation.
- **Update Function:** The aggregated information is transformed using a trainable weight matrix and a non-linear activation function.

Formally, the *update rule* for node representations in a GCN is given by:

$$h_u^{(l+1)} = \sigma\left(\sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{1}{c_u} W^{(l)} h_v^{(l)}\right),$$

where:

- $h_u^{(l)}$: *Node representation* u at layer l.
- $W^{(l)}$: *Learnable weight matrix* at layer l, responsible for feature transformation.
- c_u: Normalization constant (typically $c_u = |\mathcal{N}(u)| + 1$ for mean aggregation).
- σ: A *non-linear activation function* such as ReLU or sigmoid.

Unlike standard Graph Neural Networks, GCNs incorporate spectral graph convolutions which are derived from the eigen-decomposition of the graph Laplacian. In practice, the computational cost of such operations is reduced by approximating convolutions using first-order Chebyshev polynomials [56].

By stacking multiple layers, GCNs can propagate information further, enabling each node's representation to capture both local and global graph structure. The inclusion of self-loops in the aggregation ensures that each node retains its own information while learning from its neighbors.

GCNs have been successfully applied to combinatorial optimization (CO) problems, leveraging graph structures to efficiently find high-quality solutions. Traditional CO solvers rely on handcrafted heuristics or exact methods, which often struggle to scale to large graphs. By embedding problem-specific structures into learned representations, GCN-based solvers provide a balance between accuracy and computational efficiency, making them ideal for NP-hard problems where exact solutions are computationally infeasible.

3.4 Randomized Approximation for Multi-Way Max-Cut

A well-known randomized approximation algorithm for the *multi*-way max-cut problem assigns each node to one of the k partitions uniformly at random. This approach guarantees an approximation ratio of $1 - \frac{1}{k}$ [10, 8].

Approximation Ratio: For a given undirected graph $G = (V, E)$ with weights $w : E \to \mathbb{R}^+$, if each node is independently placed into one of the k partitions with equal probability $\frac{1}{k}$, the expected weight of edges crossing the partitions is at least:

$$\text{ratio} = \left(1 - \frac{1}{k}\right),$$

This randomized baseline is crucial for evaluating the performance of more sophisticated algorithms, including learning-based methods, by providing a straightforward benchmark.

3.5 Integer Solver (CPLEX)

IBM ILOG CPLEX Optimization Studio is a widely used solver for combinatorial optimization problems, including multi-way max-cut and balanced multi-way max-cut. CPLEX formulates these problems as Integer Linear Programming (ILP) models by introducing binary decision variables that represent node-to-partition assignments. For the multi-way max-cut problem, CPLEX aims to maximize the sum of edge weights between different partitions, while ensuring each node is assigned to exactly one partition. For the balanced version, additional constraints enforce equal partition sizes [57].

CPLEX uses advanced algorithms such as branch-and-bound, branch-and-cut, and cutting planes to explore the solution space efficiently[57]. Despite these techniques, solving large instances remains computationally intensive due to the NP-hard nature of the problem. The solver's runtime and memory requirements grow significantly with the graph size, making it impractical for very large graphs without considerable computational resources. Nonetheless, CPLEX provides exact solutions and serves as a benchmark for evaluating heuristic and machine learning-based approaches[57].

4 Methodology

In this section, we describe our neural network architecture, training process, loss functions, and post-processing techniques used in our approach to solve the *multi*-way max-cut problems.

4.1 Graph Convolutional Network (GCN) Architecture

The neural network used in this work is a two-layer Graph Convolutional Network (GCN) designed to learn node representations by iteratively propagating and aggregating information from neighboring nodes. The GCN model is implemented using PyTorch's GraphConv layers, with a softmax activation function applied in the final layer to generate probabilistic assignments of nodes to partitions.

4.1.1 Model Architecture

The Graph Convolutional Network (GCN) consists of two graph convolutional layers, followed by non-linear activation and dropout regularization. The input to the network is an adjacency matrix A of size $n \times n$, where n is the number of nodes in the graph. The value at position (i, j) in this matrix represents the edge weight between nodes i and j. The network outputs a partition probability distribution for each node, determining its assignment to one of the available partitions.

- **First Graph Convolutional Layer:** The first layer processes the adjacency matrix A and performs convolution using the Graph Convolutional operator. It transforms the input from an $n \times n$ adjacency matrix into an intermediate representation $H^{(1)}$ of size $n \times h$, where h is the hidden layer dimension. This layer captures local node interactions by aggregating neighborhood information.

- **Non-Linearity (ReLU Activation):** A ReLU (Rectified Linear Unit) activation function is applied element-wise to $H^{(1)}$ after the first convolutional layer:

  $$H^{(1)} = \text{ReLU}(H^{(1)})$$

  This introduces non-linearity, ensuring the network can learn more complex relationships rather than just linear transformations.

- **Dropout Regularization:** To mitigate overfitting and enhance the generalization capability of the Graph Convolutional Network (GCN), we incorporate *dropout regularization* in the hidden layers. Dropout is a technique that randomly sets a fraction of activations to zero during training, preventing the model from over-relying on specific features. Formally, dropout is applied to the hidden representation $H^{(1)}$ as follows:

  $$H^{(1)} = \text{Dropout}(H^{(1)}, t)$$

  where t is the dropout rate that determines the fraction of neurons to be deactivated.

  By randomly deactivating neurons, dropout helps the model learn more general patterns instead of memorizing specific details from the training data, reducing the risk of overfitting. This is particularly beneficial in GCNs, as it prevents nodes from relying too much on highly correlated neighbor information. As a result, dropout improves the model's robustness and enhances its ability to generalize to unseen data.

- **Second Graph Convolutional Layer:** The output from the first layer is passed through a second graph convolutional layer, producing a new representation $H^{(2)}$ of size $n \times k$, where k is the number of partitions. This layer refines the node embeddings, ensuring each node's features are influenced by *multi-hop neighbors*.

- **Softmax Activation (Partition Assignment):** The final output of the second convolutional layer undergoes a softmax activation function across the partition dimension:

  $$P_i = \text{softmax}(H_i^{(2)})$$

  This ensures that each node i receives a probability distribution over the k partitions, with values summing to 1. The softmax function helps in probabilistic node classification, allowing post-processing methods to extract discrete assignments.

The loss function is optimized using stochastic gradient descent (SGD) with backpropagation. The *final node representations* obtained from the GCN are used for partitioning the graph into multiple disjoint subsets as explained below, optimizing for a multi-way max-cut problem. The probabilistic assignments generated by softmax allow for flexible decision-making, which is further refined using post-processing heuristics.

4.2 One-Hot Encoding of Output

To ensure that each node is assigned to a single partition, we apply a one-hot encoding operation to the GCN's probabilistic output before computing the loss. This transformation converts the softmax probability distribution into a categorical representation, where each node is assigned exclusively to one partition. The one-hot encoding is defined as:

$$y_i = \begin{cases} 1 & \text{if } i = \arg\max(p) \\ 0 & \text{otherwise} \end{cases}$$

where:

- p is the softmax output vector representing the probability distribution over partitions.
- i indexes the partitions.
- $\arg\max(p)$ returns the index of the partition with the highest probability.
- y_i is the one-hot encoded assignment of the node to a specific partition. [CHANGE]

A one-hot vector is a binary vector in which exactly one element is set to 1, and all other elements are set to 0. This ensures that each node is assigned to a single partition, enforcing a discrete selection rather than a probabilistic weighting across multiple partitions. We also ensure that we assign the terminals to the correct partitions everytime we compute a one-hot vector function on the neural network output.

By applying one-hot encoding before computing the loss, we prevent the neural network from expending unnecessary computational effort in fine-tuning probabilities toward 0 or 1. Without this step, the model might prioritize reducing entropy in the probability distribution rather than optimizing the actual max-cut objective. This could lead to the network focusing on making its outputs resemble deterministic assignments instead of improving partition quality.

Furthermore, the one-hot encoding mechanism preserves the ability to compute gradients for backpropagation. Instead of allowing the neural network to refine probability values, the optimization process focuses on maximizing the cost of the solution, leading to more effective training.

4.3 Loss Functions

The objective function for the *multi*-way max-cut problem is to maximize the total weight of the edges that are removed. Hence, we define the following loss function, inspired by a Hamiltonian partition formulation [12]:

$$\mathcal{L}_{\mathcal{C}} = -\frac{1}{2} \sum_{i,j \in V} A_{ij} \cdot (1 - s_i \cdot s_j^T),$$

where:

- $\mathcal{L}_{\mathcal{C}}$ is the loss function that we aim to minimize, ensuring that edges with higher weights have endpoints in different partitions.
- V is the set of nodes in the graph.
- A_{ij} is the adjacency matrix entry representing the weight of the edge between nodes i and j. If no edge exists between nodes i and j, then $A_{ij} = 0$.
- s_i is the one-hot encoded partition assignment vector for node i, where only one element is set to 1, indicating the partition to which node i belongs, while all others elements are set to 0.
- s_j^T is the transposed one-hot vector for node j.
- $1 - s_i \cdot s_j^T$ ensures that an edge contributes positively to the loss only if tits endpoints i and j are assigned to different partitions. If both endpoints are in the same partition, this term evaluates to zero, meaning the edge does not contribute to the cost of the multi-cut.
- The factor $\frac{1}{2}$ is included to correct for double-counting, as the adjacency matrix A is symmetric, meaning each edge is considered twice in the summation.

By minimizing this loss function, the neural network is trained to maximize the weight of edges with endpoints in different partitions, leading to an optimal multi-way max-cut. The use of one-hot encoded vectors ensures that the model directly learns discrete partition assignments rather than probability distributions. This formulation effectively guides the neural network toward producing high-quality partitions while maintaining computational efficiency.

In the balanced *multi*-way max-cut problem, we aim not only to maximize the total weight of edges between partitions but also to enforce that each partition contains an approximately equal number of nodes. To achieve this, we modify the loss function by incorporating an additional balance constraint. We introduce $\mathcal{L}_B$:

$$\mathcal{L}_B = \sum_{k=1}^{K} \left(\sum_{v \in V} s_{vk} - \frac{|V|}{K} \right)^2.$$

This term ensures that each partition contains an approximately equal number of nodes.

- K is the number of partitions.

- s_{vk} is the one-hot encoded assignment of node v to partition k, meaning $s_{vk} = 1$ if node v is in partition k and 0 otherwise.
- $\sum_{v \in V} s_{vk}$ counts the total number of nodes assigned to partition k.
- $\frac{|V|}{K}$ is the ideal number of nodes per partition in a perfectly balanced partitioning.
- The squared term $\left(\sum_{v \in V} s_{vk} - \frac{|V|}{K}\right)^2$ penalizes deviations from perfect balance.

By combining $\mathcal{L}_\mathcal{C}$ and $\mathcal{L}_B$, we create a loss function that allows us balanced paritioning for multi-way max-cut:

$$\mathcal{L} = \mathcal{L}_\mathcal{C} + \lambda \mathcal{L}_B,$$

where:

- $\mathcal{L}_\mathcal{C}$ is the Hamiltonian-based max-cut loss.
- λ is a hyperparameter that controls the trade-off between optimizing the cut value and maintaining balanced partitions.
- $\mathcal{L}_B$ is the balance constraint that enforces approximately equal partition sizes.

By minimizing $\mathcal{L}_B$, the model is encouraged to distribute nodes as evenly as possible among the partitions. The hyperparameter λ is used to adjust the relative importance of partition balance versus maximizing the cut value.

4.4 Generating Training and Testing Graphs

To evaluate the performance of our GCN model, we generated a diverse set of training and testing graphs with varying sizes and structures. This section outlines the process for graph generation, terminal assignment, and dataset preparation.

Graph Generation: Graphs were generated as d-regular graphs using the NetworkX library, ensuring each node has the same degree. Node degrees were set within specific ranges based on the experiment, and graph sizes varied between predefined intervals. This variability facilitated the assessment of the GCN's scalability and generalization across different graph sizes.

Terminal Nodes: For each graph, k unique terminal nodes were randomly selected and swapped with nodes labeled 0 to $k-1$ (e.g., nodes 0, 1, and 2 for a 3-way max-cut). This ensured that each terminal was assigned to a distinct partition, simplifying the neural network's partitioning process and maintaining consistency across all graphs.

Adjacency Matrix and Padding: Each graph was represented by an adjacency matrix with edge weights set to 1. Smaller graphs were padded with zeros to match the GCN's required input size, ensuring uniform input dimensions during training.

Final Dataset: The final dataset was stored as a dictionary, with each entry containing the DGL graph, padded adjacency matrix, original NetworkX graph, and terminal nodes. Multiple datasets with varying sizes and degrees were generated, providing structured inputs for the GCN model to learn and evaluate partitioning solutions efficiently.

4.5 Training Procedure

The training process utilizes datasets consisting each of a group of graphs, each consisting of an adjacency matrix and a set of terminal nodes.

1. **Pass the training graph and input features through the GCN**: Each input graph is provided as a DGL graph structure G and its corresponding adjacency matrix A, which serves as the node features. The GCN processes this graph through two layers of graph convolution, capturing both local and global graph structures. The first layer generates intermediate embeddings, followed by a ReLU activation for non-linearity, and the second layer outputs final node embeddings representing probabilistic partition assignments.

2. **Apply one-hot encoding to enforce discrete partition assignments**: After the forward pass, the model's output consists of probability distributions over partitions for each node. A one-hot encoding operation is applied by selecting the partition with the highest probability for each node, ensuring discrete assignments. Additionally, terminal nodes are fixed in their designated partitions throughout training to maintain partition validity.

3. **Compute the loss**: The loss is computed using the defined max-cut objective inspired by the Hamiltonian partition formulation(as described in section 4.3). The loss function quantifies the quality of the partitioning by considering both the edge cut value (to maximize separation between partitions) and the balance constraint (for the balanced version of the problem to ensure partitions have similar sizes. A lower loss indicates more optimal partitions.

4. **Perform backpropagation to update model parameters**: The loss is propagated backward through the network to compute gradients for all learnable parameters. Using the optimizer, these parameters are updated iteratively to minimize the loss, improving the model's partitioning ability over time.

To enhance generalization and prevent overfitting, the training method incorporates early stopping based on *cumulative loss*. The *cumulative loss* refers to the running sum of the model's loss over multiple epochs, rather than just considering the loss from a single iteration. It provides a more stable measure of training progress by smoothing out fluctuations from individual batches, helping determine whether the model is genuinely improving or merely experiencing short-term variations. If the cumulative loss does not show significant improvement over a predefined number of epochs, training is halted to prevent unnecessary computation and overfitting.

This early stopping mechanism ensures that the model does not continue to optimize minor fluctuations or noise in the data rather than capturing meaningful graph structures. Without early stopping, the model might begin fitting to spurious correlations present in the training set, leading to reduced generalization performance on unseen graphs. By halting training at the optimal point, we prevent the model from learning irrelevant patterns that do not contribute to the overall objective.

Additionally, the model state corresponding to the best loss value is saved during training. Since training involves stochastic updates, the model may reach a better partitioning solution before early stopping is triggered. By saving the best-performing model state, we ensure that the final model used for inference corresponds to the most optimal partitioning found during training.

4.6 Post-Processing

To refine the output of the Graph Convolutional Network (GCN), we employ a post-processing heuristic that probabilistically assigns nodes to partitions and iteratively searches for the best partitioning. This additional step improves partition quality by maximizing cost of the solution obtained from the GCN's probabilistic output. We apply different post-processing techniques for multi-way max-cut problem and balanced multi-way max-cut problem.

4.6.1 Multi-way Max-cut Post-processing

The algorithm for post-processing of multi-way max-cut is formally defined as follows:

Post-Processing Algorithm: Let n denote the number of iterations in the refinement process. The algorithm iterates over n candidate solutions to find the best partitioning. For each iteration $i = 1$ to n, where is n is $MIN(\text{number\_of\_nodes}, 1000)$ the following steps are performed:

1. **Generate a partition assignment:** Each node u is assigned to a partition based on the GCN output probabilities. Let $\mathbf{p}_u = [p_{u,1}, p_{u,2}, \ldots, p_{u,k}]$ represent the softmax probability vector for node u, where $p_{u,k}$ is the probability of node u belonging to partition k. The partition assignment s_u is determined by drawing a random number $r \sim U(0,1)$ and selecting the partition index k as follows:

Find the best value that improves solution, but does not take too long.

$$s_u = \arg\min_k \left(\sum_{j=1}^{k} p_{u,j} \geq r \right).$$

Here:

- $\mathbf{p}_u$ is the probability vector for node u, obtained from the GCN output.
- $r \sim U(0, 1)$ is a randomly drawn value from a uniform distribution between 0 and 1.
- s_u is the selected partition index based on probabilistic sampling.

This approach ensures that nodes are assigned in a probabilistic manner, preserving the uncertainty captured by the GCN while introducing randomness that allows for exploration of alternative partitioning solutions.

2. **Compute the max-cut objective value:** The partition assignment $\mathbf{s}$ is evaluated using the max-cut objective function:

$$f(\mathbf{s}) = \sum_{(u,v)\in E} w_{uv} \cdot \mathbb{I}[s_u \neq s_v],$$

where:

- $f(\mathbf{s})$ is the computed max-cut value for the current partition assignment.
- s_u and s_v are the partition assignments of nodes u and v.
- $\mathbb{I}[s_u \neq s_v]$ is an indicator function that evaluates to 1 if nodes u and v belong to different partitions, and 0 otherwise.

This function measures the total weight of edges crossing different partitions, which is the objective function in multi-way max-cut optimization.

3. **Update the best partition assignment:** If the current partition assignment $\mathbf{s}$ results in a higher solution value than the previously best-known solution f^*, update the best partition and score:

$$f^* = \max(f^*, f(\mathbf{s})),$$
$$\mathbf{s}^* = \mathbf{s} \quad \text{if } f(\mathbf{s}) > f^*.$$

Here:

- f^* represents the best max-cut value found so far.
- $\mathbf{s}^*$ stores the partition assignment corresponding to the highest observed solution value.

This step ensures that across multiple iterations, the partitioning with the highest value is retained as the final output.

The final partition assignment $\mathbf{s}^*$ obtained after n iterations is used as the refined output of the GCN model. By iteratively refining the solution, this post-processing step enhances the quality of the final graph partitioning, ensuring that the neural network's probabilistic predictions translate into a high quality multi-way max-cut solution.

Heuristic Example To illustrate how the heuristic operates, consider a node with predicted probabilities for three partitions: [0.2, 0.7, 0.1]. The algorithm performs the following steps:

- Generate a random number between 0 and 1. If, for example, the number is 0.1, it falls within the range of the first partition ($p_1 > 0.1$).
- If the random number is 0.5, it falls within the range of the second partition ($p_1 + p_2 > 0.5 > p_1$).
- If the random number is 0.95, it falls within the range of the third partition ($p_1 + p_2 + p_3 > 0.95$).

4.6.2 Balanced Multi-way Max-cut Post-processing

The post-processing algorithm for the balanced multi-way max-cut problem is as follows:

1. **Determine Partition Sizes and Target:** First, compute the number of nodes in each partition in the solution obtained from the neural network. Define the target size for each partition as the total number of nodes divided by the number of partitions.

2. **Identify Imbalance:** Next, calculate the difference between the current size of each partition and the target size. Partitions with more nodes than the target are labeled as excess partitions, while those with fewer nodes are identified as deficient partitions. This step indicates which partitions require node removal and which need additional nodes.

3. **Select Nodes for Transfer from Excess Partitions:** For each excess partition, examine every node by counting the number of neighbors within the same partition. Rank the nodes in descending order according to this in-partition connectivity. Nodes with higher connectivity are ideal candidates for transfer, as moving them will likely increase the overall max-cut value. Select the appropriate number of top-ranked nodes from each excess partition, corresponding to the number needed to reach the target size.

4. **Assign Nodes to Deficient Partitions:** Determine how many nodes each deficient partition requires to reach the target size. Reassign the selected nodes from the excess partitions to the deficient ones. This assignment is randomized into deficient partitions and moved around for $(\min(number\_of\_nodes, 1000)$ of times, ensuring that each deficient partition receives exactly the number of nodes required to balance it and getting optimal max-cut.

5. **Finalize and Validate:** Verify that all partitions have been adjusted to match the target size. Then, compute the overall max-cut value to ensure that the node reassignments have maintained or improved the cut quality. This final validation step confirms that the balance constraint has been met without compromising the value of the solution.

5 Experimental Results

We conducted extensive numerical experiments to evaluate the performance of our proposed Graph Convolutional Neural Network (GCN) model on both the multi-way max-cut and balanced multi-way max-cut problems. Our evaluation compared our algorithm with the randomized algorithm described in Section 3.4—one of the best known approximation ratios for multi-way max-cut—as well as with the CPLEX solver. Although some semi-definite programming–based methods offer slightly better approximation guarantees, ~~but~~ solvers are too slow for the graph sizes used in our experiments. To ensure consistency, the GCN architecture was maintained in a manner similar to that of Schuetz et al. [1] (as detailed in the Methodology section).

All experiments were implemented using the PyTorch library ~~}~~version 2.2.0.post100~~}~~, with ~~the~~ `GraphConv` ~~layer~~ as a key component for graph convolution. Neural network training was performed on systems selected based on graph size: for graphs with fewer than 2000 nodes, a machine with 128 GB of RAM was used, while for graphs with 2000 to 10,000 nodes, a system with 256 GB of RAM was employed. The experiments were executed on a CPU setup consisting of 2 $\times$ Intel Gold 6148 Skylake processors at 2.4 GHz, utilizing eight active cores and supported by 480 GB of SSD storage. In contrast, the CPLEX solver was run on a system with 256 GB of RAM for all graph sizes ~~and cut configurations,~~ except for experiments with $k = 100$, where CPLEX required 350 GB of RAM.

~~We compared our GCN's performance against both the randomized algorithm and CPLEX~~. In our experiments, CPLEX was allotted a maximum of one hour per test instance, and thus it did not always converge to an optimal solution. Accuracy is defined as the ratio of the solution value produced by an algorithm to that produced by CPLEX; given that CPLEX may time out on larger graphs, this ratio represents an estimate of the true approximation ratio. Standard deviation values for each ~~result~~ are also provided.

5.1 Accuracy & Performance

5.1.1 Multi-Way Max-Cut

We performed extensive experiments to measure the performance of our GCN for the multi-way max-cut problem, for instance where k=3, 4, 5, 10, and 100. The maximum k-way cut problem has application in network design, scheduling and VLSI design, among others. These applications often involves small values of k[58, 59].

Experiment A: Fourteen unique training sets were created, each consisting of 340 graphs with sizes ranging from 200 to 800 nodes and degrees from 6 to 12. A separate GCN model was trained for each of these training sets, ensuring that each neural network experienced a distinct training environment. To evaluate scalability and generalization, each model was tested on graphs ~~tested~~ with up to 8000 nodes (10 times larger than the maximum training size). The testing set included 20 graphs for each size range, and the results are summarized in Figures 1 and ~~Figures~~ 2. Each green, orange and red bar in the figures is labelled with a percentage, which is the ratio of the mean value of the solutions computed by the corresponding algorithm to the mean value of the solution computed by CPLEX. The black interval above represents the variance.

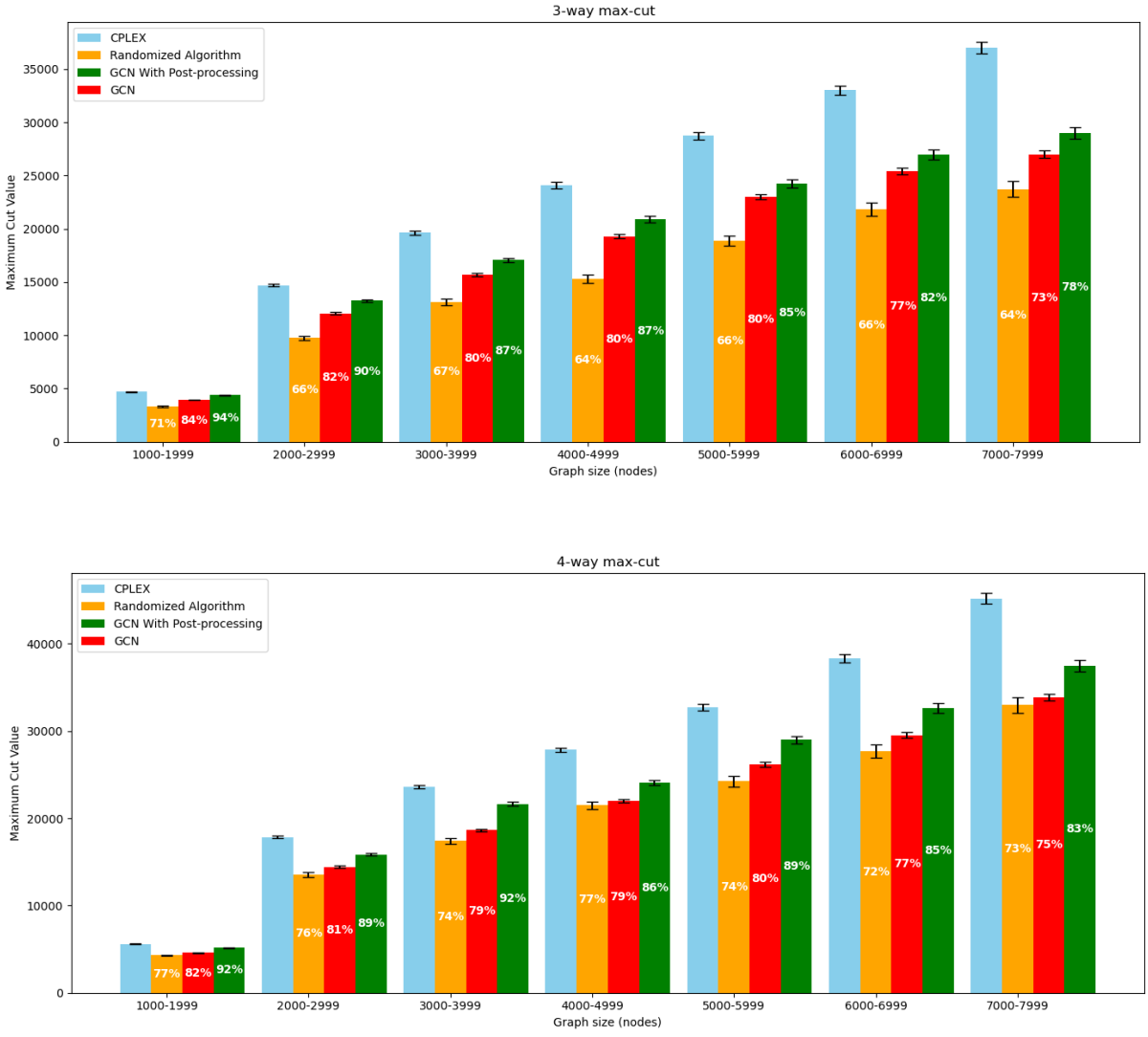


Figure 1: A comparison of the accuracy of algorithms for 3-way Maxcut, and 4-way Maxcut. The percentage indicates accuracy relative to ~~integer solver~~ CPLEX.

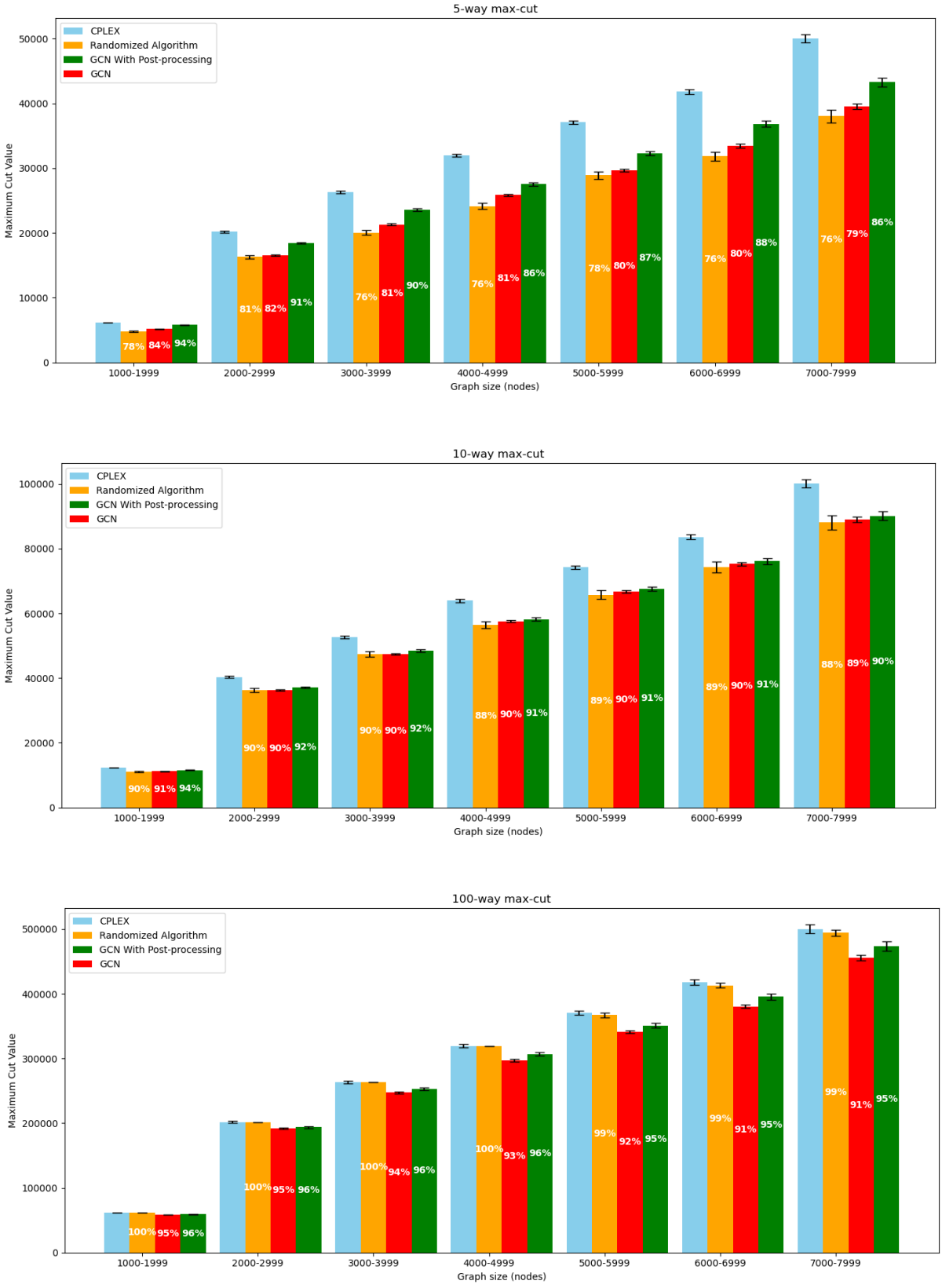
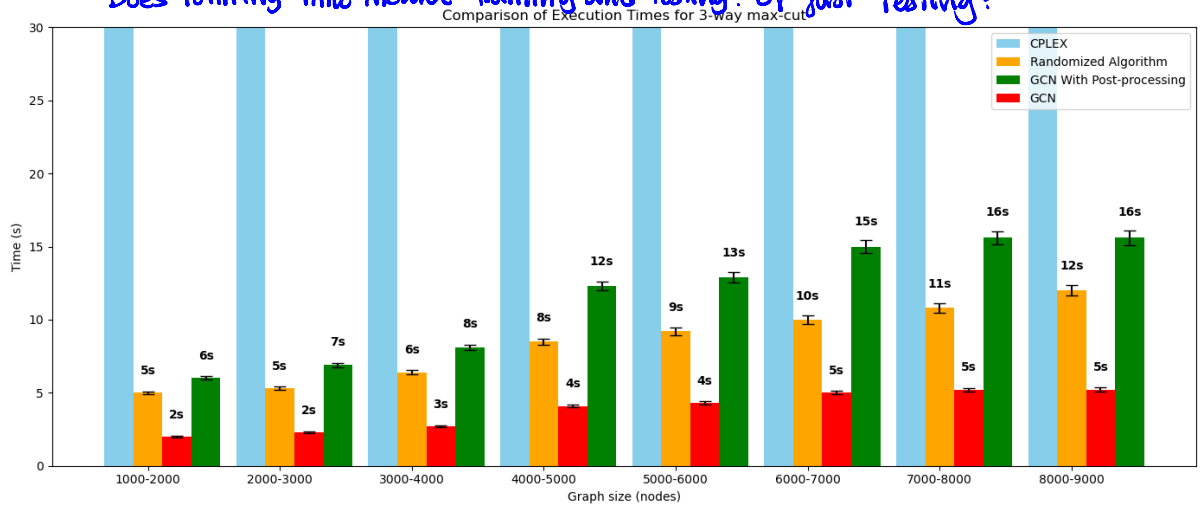


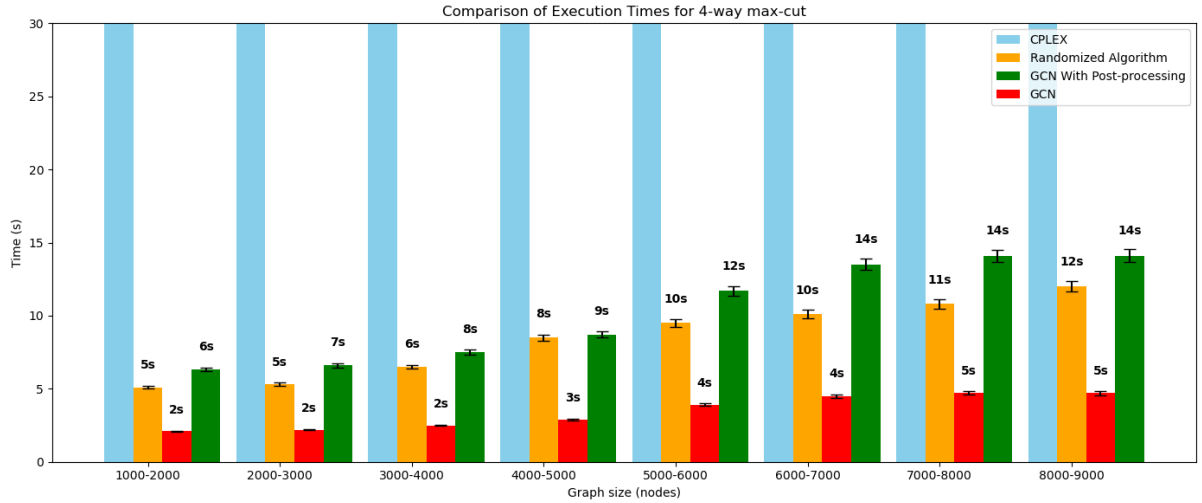
Figure 2: A comparison of the accuracy of algorithms for 5-way Maxcut, 10-way Maxcut and 100-Maxcut.

Let's have a look at performance for our GCN with randomized algorithm. The Figure 3 only includes testing performance. Figures 3 and 4 show running times of the algorithms. The running time of CPLEX was one hour for each test.

- Post processing takes longer than GCN for just a small improvement. Why is post processing so slow if it is very simple?

- Does running time include training and testing? Or just testing?





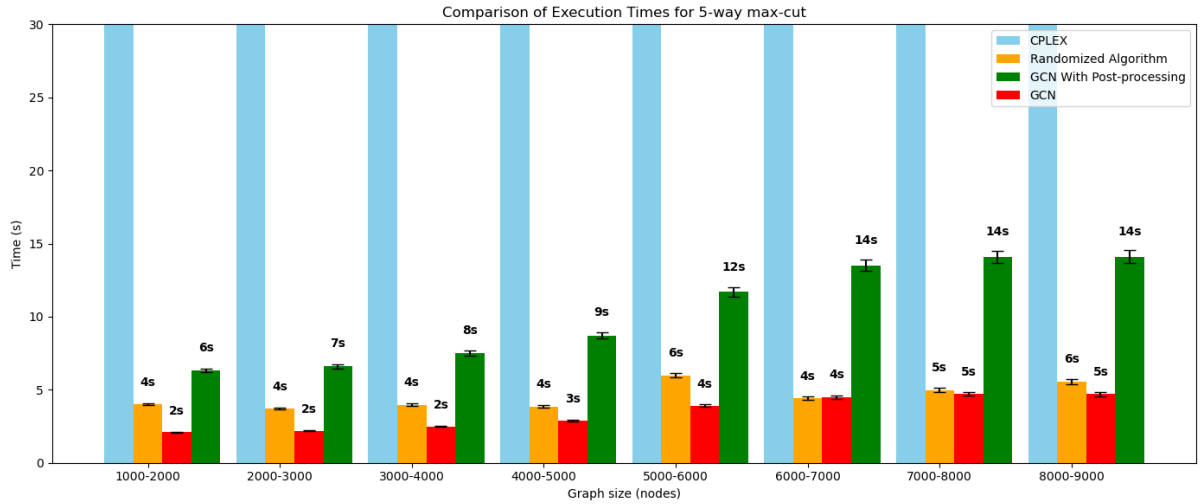
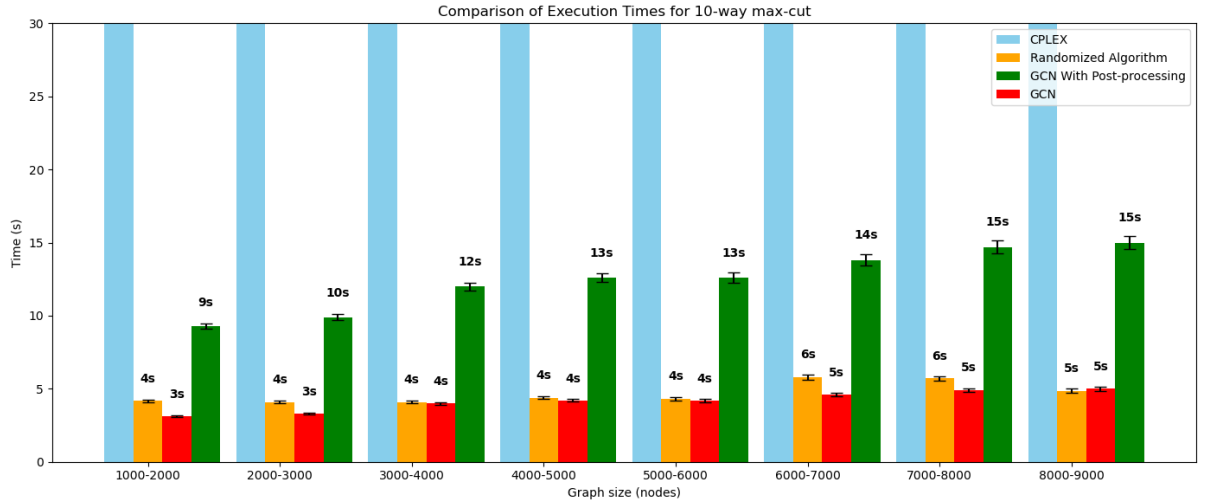


Figure 3: A comparison of the running times of our 3-way, 4-way and 5-way max-cut ~~GCN, randomized algorithm and CPLEX.~~ the algorithms for Note that CPLEX ran for one hour (3600s) for each test.



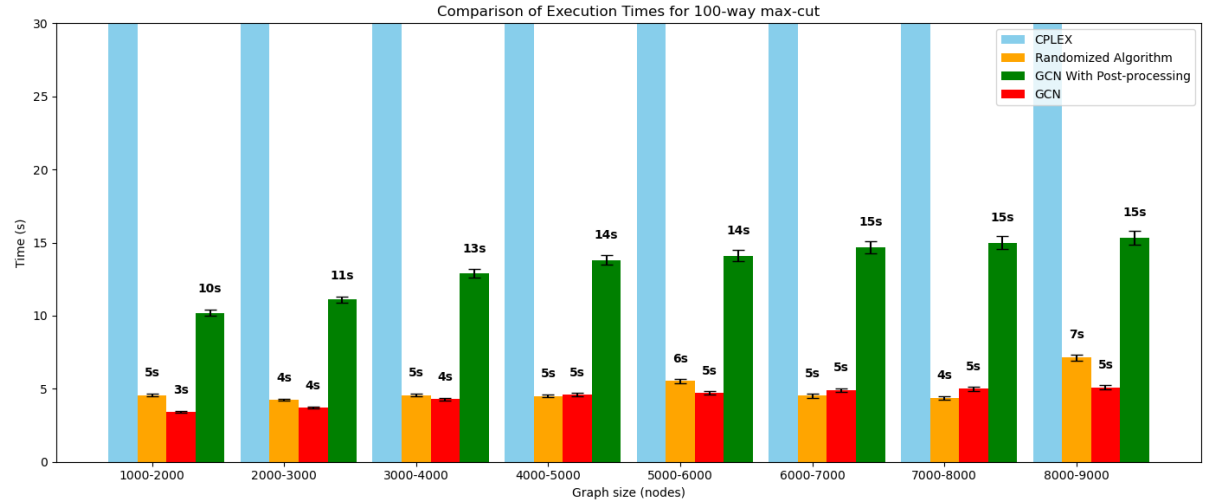
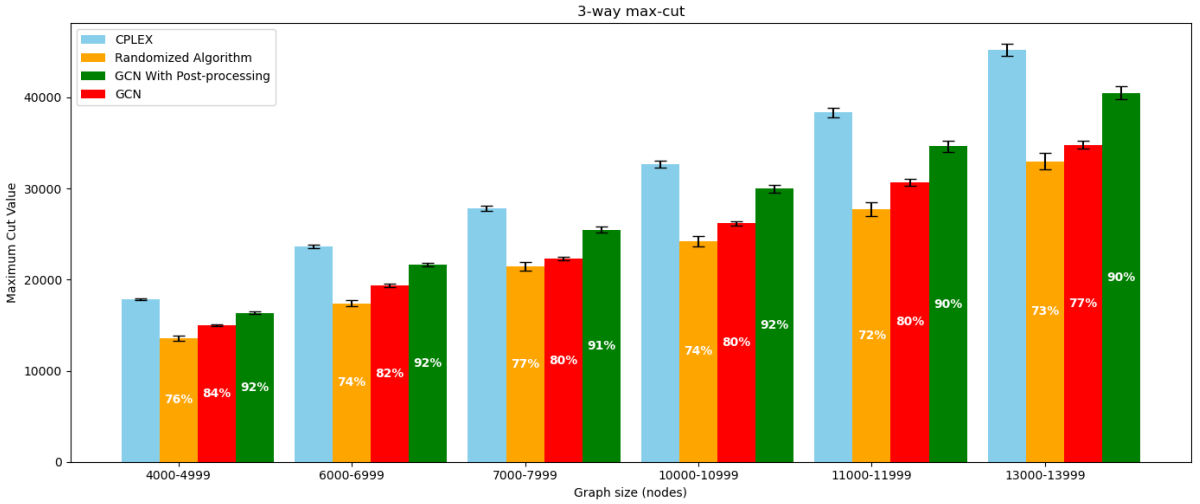
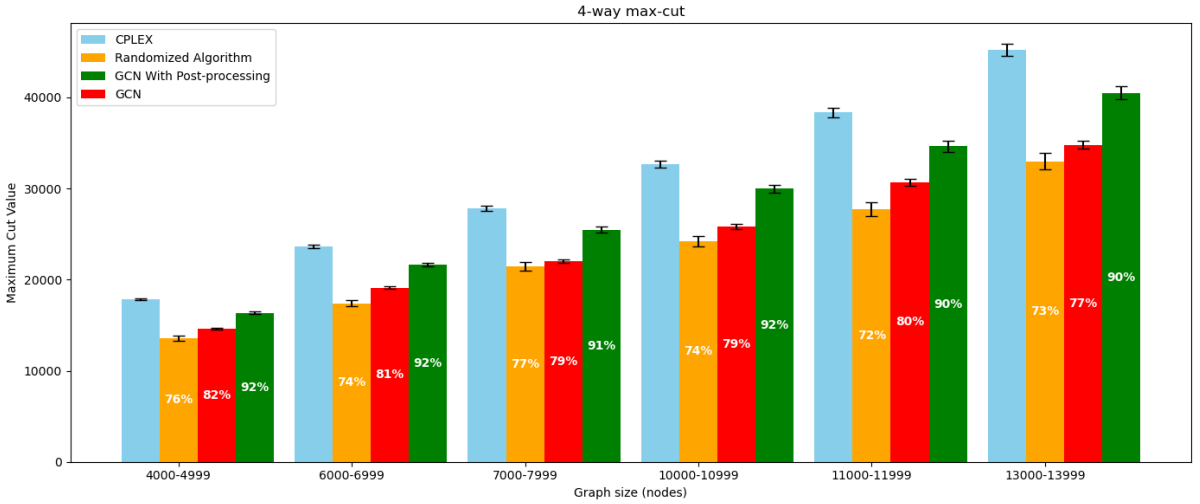


Figure 4: A comparison of the running times of ~~our~~ the algorithms for 10-way, and 100-way ~~max-cut GCN, randomized algorithm and CPLEX~~. Note that CPLEX ran for one hour (3600s) for each test.

Experiment B: Five training sets were created, each containing 340 graphs with sizes ranging from 5000 to 8000 nodes and degrees from 6 to 12. A separate GCN model was trained on each training set, ensuring that every neural network was trained on a distinct set of graphs. For testing, 20 graphs from each size range were used, and the results are reported in Figures 5.





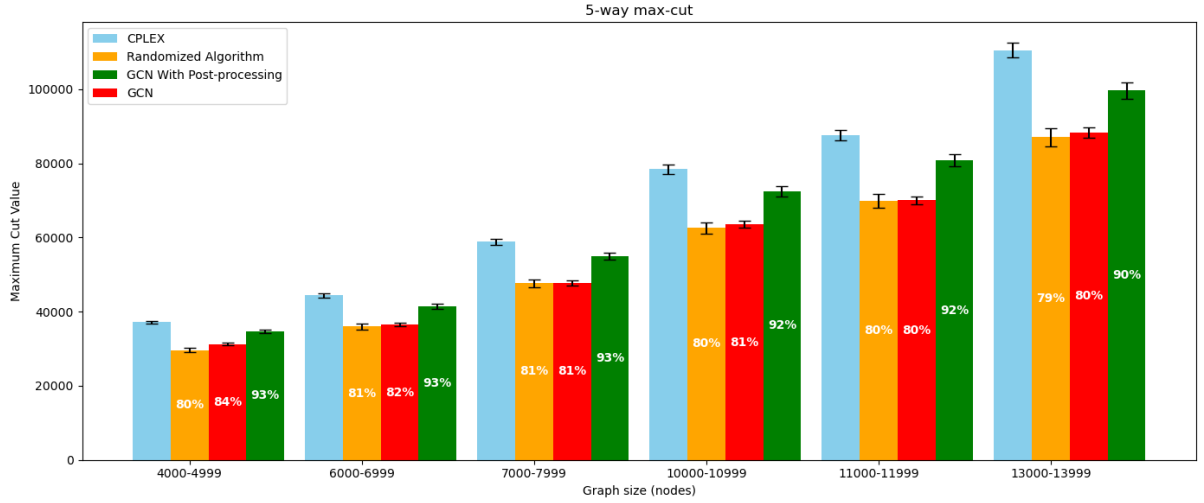


Figure 5: A comparison of the accuracy of algorithms for 3-way Maxcut, 4-way Maxcut, and 5-way Maxcut for larger graphs size (ranging from 6000 nodes - 13500 nodes). The percentage indicates accuracy relative to integer solver.

why not show running time ?

Comparison Setup and Baselines In this work, we evaluate two baseline methods—a **CPLEX** and a **randomized** algorithm—against our proposed **GCN** and **GCN with post-processing** approaches. The CPLEX solver is allotted

a maximum runtime of one hour per instance. Consequently, it may not always reach the true optimum, but it still provides a strong upper bound on solution quality. The randomized algorithm, drawn from section 3.4, serves as a simple but relatively fast baseline that assigns partitions. Repetitive.

Accuracy Observations Across all experiments (for different partition cardinalities and varying graph sizes), CPLEX consistently yields the highest cut values within its allotted time. However, our GCN based methods approach or closely match CPLEX's solution quality in many cases (shown in figure 1,2 and 5), especially on mid-sized problems (up to around 5 000–8 000 nodes as shown in figure 5). For larger problems, if CPLEX fails to converge to the global optimum within one hour, the performance gap between the GCN approaches and the solver often decreases. The variance bars in the figures 1, 2 and 5 are relatively small for the GCN methods. This indicates that, once trained, the GCN inference is stable across different runs.

The **GCN with post-processing** tends to outperform the standalone GCN by a modest but noticeable margin, indicating that even a short local improvement step can increase cut value. Meanwhile, the randomized baseline lags substantially behind, with observed cut values anywhere from 5% to 15% below the GCN methods (depending on the graph size and the number of partitions). In terms of consistency, the GCN methods exhibits relatively low variance (figure 1 and 2), suggesting their output is stable across different runs. The randomized baseline naturally carries higher variance due to its intrinsic randomness.

Runtime Observations Regarding runtime, CPLEX almost always hits the one-hour time limit for larger graphs, reflecting the challenge of solving integer programs at scale. In contrast, both GCN-based approaches execute in a matter of seconds, even for tens of thousands of nodes. Although the post-processing step incurs a slight overhead, it remains within the same low time complexity range as pure GCN inference. We do observe from Figure 4, that the runtime for raw GCN says consistent regardless of the number of partitions. The variance of GCN throughout runs are also small compared to other methods, indicating GCN yielding consistent runtime result once trained.
The randomized baseline, despite being conceptually simple, takes somewhat longer than raw GCN inference in many instances. Nonetheless, both randomized and GCN methods are dramatically faster than running CPLEX for an hour, making them attractive for real-time or large-scale scenarios.

The results highlight a clear trade-off: CPLEX may yield the best absolute solution when given sufficient time, yet its one-hour limit remains impractical for large instances. The GCN approaches, particularly the variant augmented by post-processing, straddle a sweet spot of high-quality cuts in mere seconds. The randomized baseline provides an additional perspective on how a naive but fast method compares to more sophisticated strategies. Ultimately, the GCN plus local refinement shows promise for scaling multi-way max-cut across broad problem sizes with near-optimal accuracy and minimal runtime.

5.1.2 Balanced Multi-way Max-cut

We ran similar experiments to test the performance of our GCN on the balanced multi-way cut problem. For balanced multi-way max-cut instances, the quality of the solutions computed by our GCN was slightly lower due to that of the solutions obtained for multi-way max-cut problems, to the additional constraints imposed by the balancing requirements. We did not use the randomized algorithm as comparison benchmark because that algorithm can not produce paritions of the same size.

Comparison: GCN vs. CPLEX

In this section, we compare the performance of our GCN with that of CPLEX on 5 versions of the problem: 3-way, 4-way, 5-way, 10-way and 100-way balanced max-cut. Figures 6 and 7 show the results for each versions.

Experiment A: Eight GCN models were trained for each balanced multi-way max-cut problem on a dataset of 340 graphs. Each model was trained on different graph sets, with sizes ranging from 200 to 800 nodes and degrees from 6 to 12. To assess scalability and generalization, we evaluated the models on graphs with up to 2400 nodes (3x the training size), this tested the model's their robustness on unseen graph sizes and topologies. The testing set contained

20 graphs for each size range. The mean average accuracy and performance is summarized in the figures 6 and 7. The line chart indicates variance.

which?

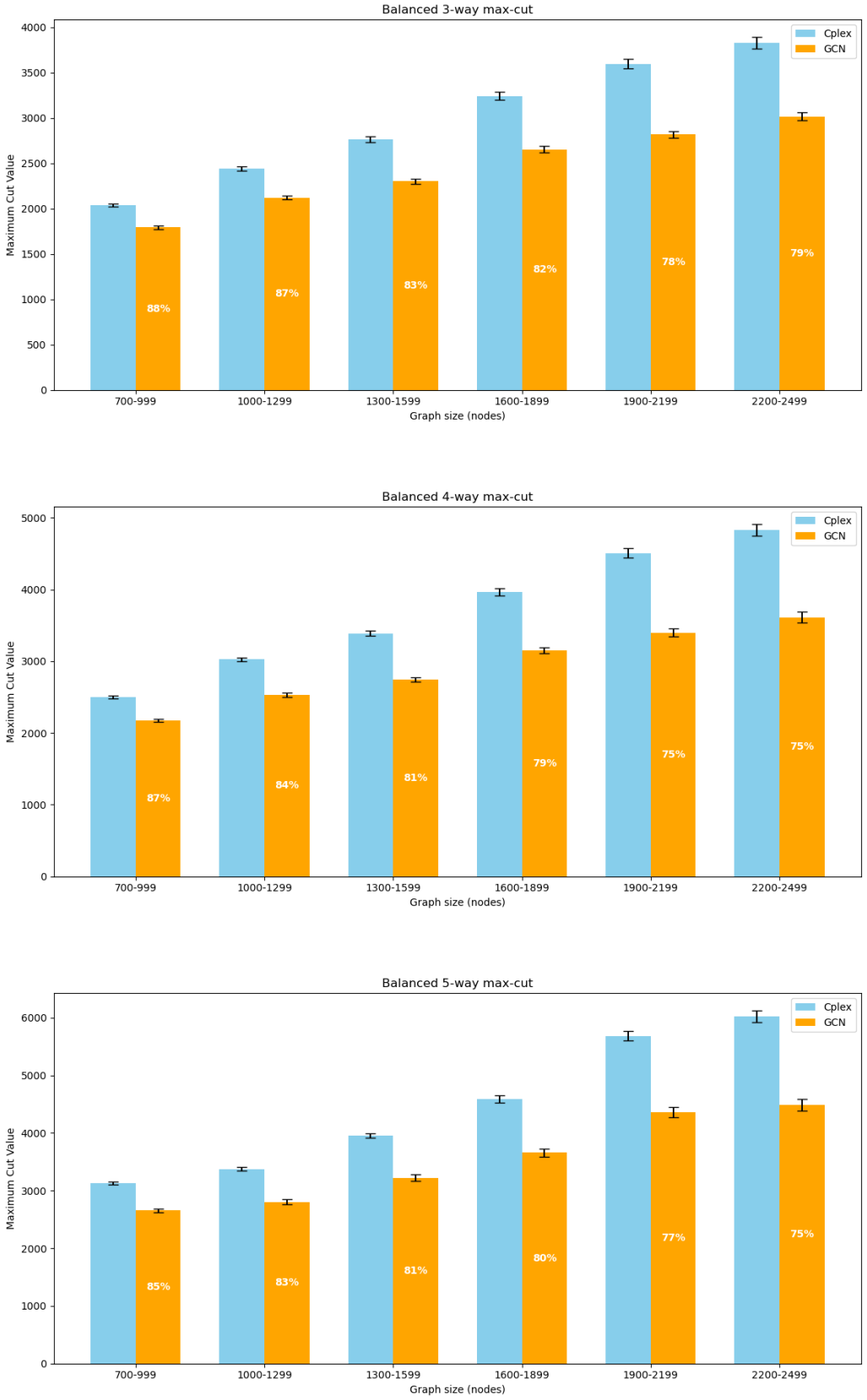
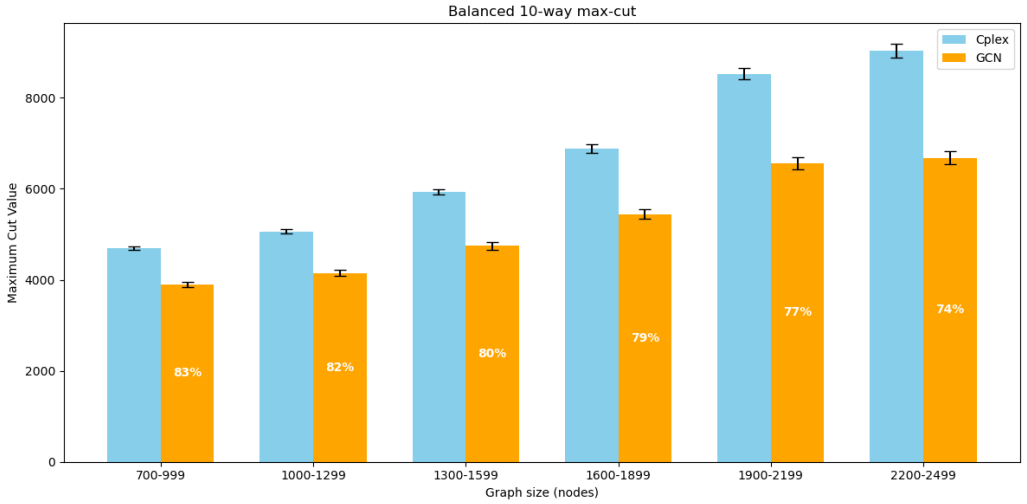


Figure 6: Accuracy of our GCN for balanced 3-way, 4-way and 5-way balanced max-cut.



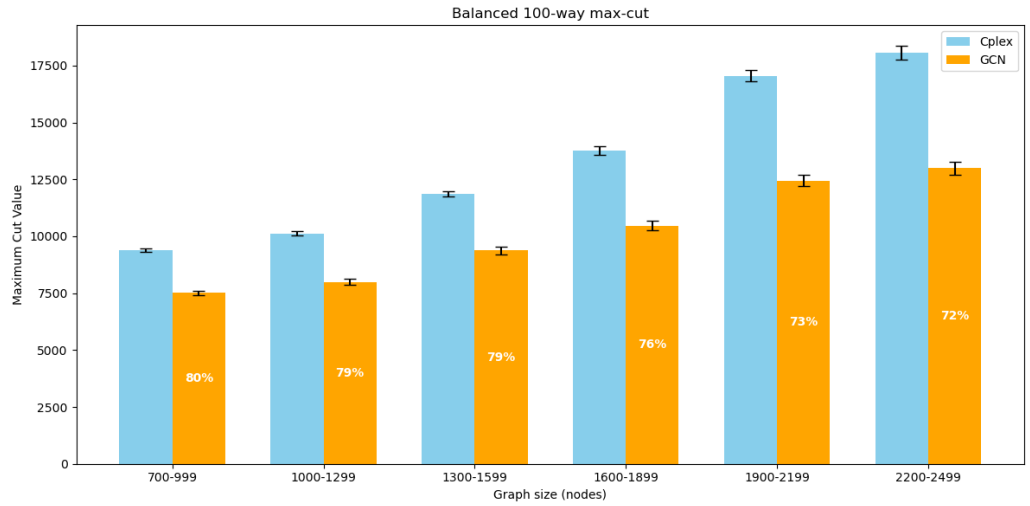
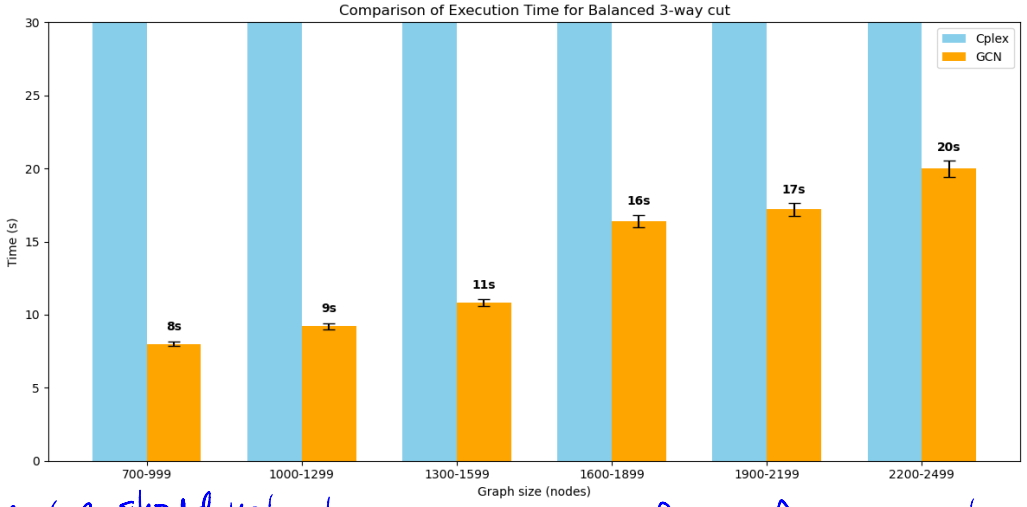


Figure 7: Accuracy of our GCN for balanced 10-way and 100-way ~~balanced~~ max-cut.

The following figures show the running time of

~~Now lets have a look at performance for~~ our GCN.



Maybe we should not show CPLEX in the figures for running times.

Figure 8: A comparison of the running times of our balanced 3-way max-cut GCN and CPLEX.

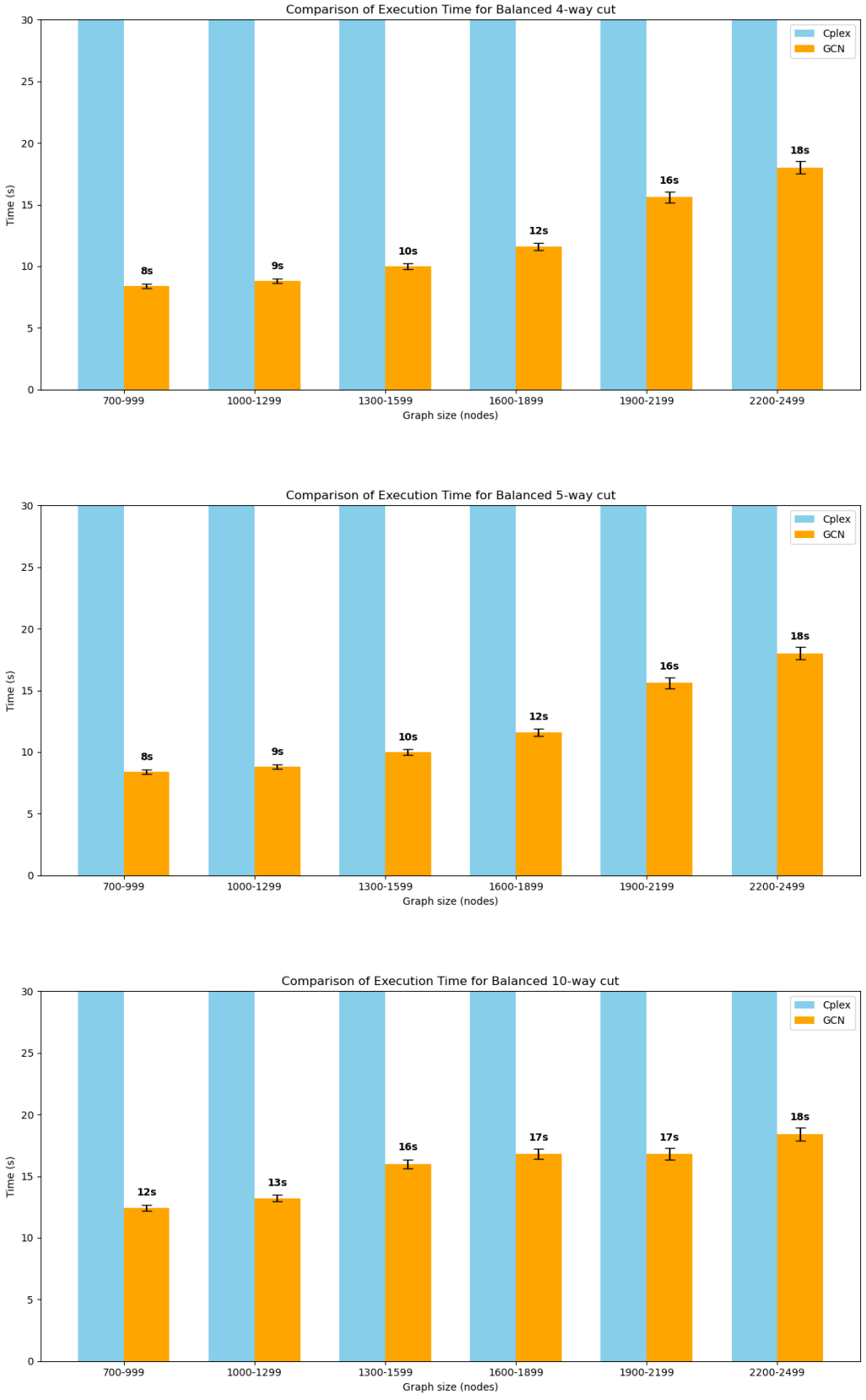


Figure 9: A comparison of the running times of our balanced 4-way, 5-way and 10-way max-cut GCN and CPLEX.

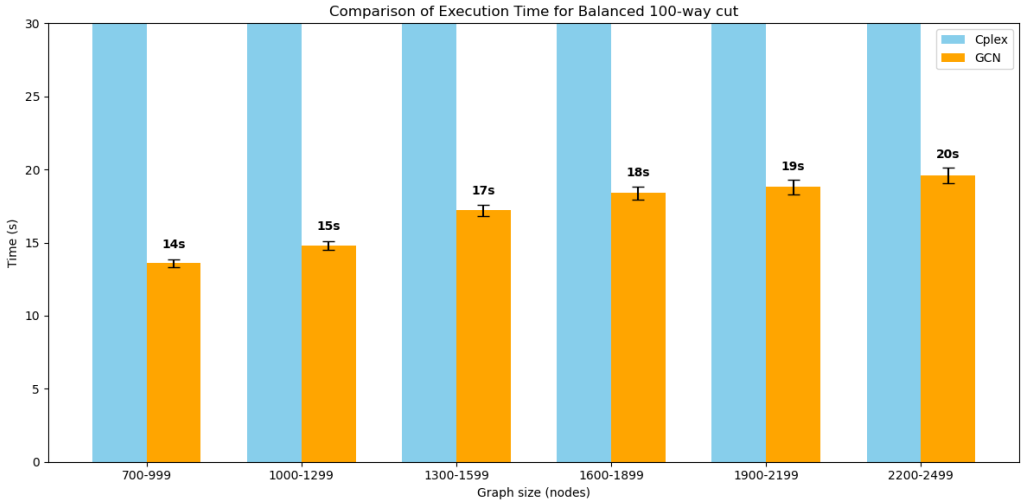


Figure 10: A comparison of the running times of our balanced 100-way max-cut GCN and CPLEX. The CPLEX ran for 1 hour (3600s)

The results, shown in figure 9, and 10 for the balanced multi-way max-cut problem indicate that our GCN-based approach consistently obtains a sizable fraction of the cut value produced by CPLEX. Notably, CPLEX is run for a maximum of one hour in each instance, so it may not always attain the absolute optimal solution, especially as the number of nodes and partitions increase. In the runtime plots, shown in figure 8 and 9, however, we truncate the CPLEX bars at 30 seconds for visual clarity. This means the depicted bars do not fully reflect the one-hour solver executions; rather, they simply show that beyond the 30-second mark, CPLEX's computation time remains significantly higher than that of our GCN.

Accuracy Observations. Our GCN solutions commonly reach around 70%–90% of CPLEX's achievable cut, demonstrating that even under the strict balancing constraint, the learned model yields robust cuts. While the solver has more time to explore the integer programming formulation, the GCN quickly produces a near-optimal solution that remains within a reasonable gap of what CPLEX finds after much longer computation.

Runtime Observations. The truncated bars at 30 seconds highlight the stark contrast in computational cost: CPLEX, which actually runs up to one hour, often fails to converge to a global optimum in that window. In many instances, the solver remains locked in its branch-and-bound search, seeking a valid balanced partition. The GCN, on the other hand, typically finishes inference in well under 30 seconds, scaling much more gracefully for the tested graph sizes. Any additional overhead from balancing constraints is negligible compared to the exhaustive nature of integer programming.

From a practical standpoint, these findings emphasize the trade-off between solution quality and time. While CPLEX may, in principle, locate a higher cut value by extensively exploring the solution space, the one-hour cap can still leave large instances partially solved or approaching suboptimal plateaus. The GCN's speed, even for high partitions, makes it appealing for real-time or large-scale scenarios where an extended solver run time is not feasible. Thus, the GCN offers a pragmatic balance of good cut quality and fast execution in solving the balanced multi-way max-cut problem.

Across all balanced max-cut configurations, our experiments reveal that our GCN-based approach produces high-quality solutions relative to CPLEX, for graphs of sizes similar to those used during training size. However, as graph size and the number of partitions increase, both accuracy and speed are adversely affected. In particular, while for the 3-way, 4-way, and 5-way balanced max-cut our GCN maintain acceptable accuracy (ranging from between 75% to 88%), for the 10-way and especially for the 100-way balanced max-cut the GCN shows a more pronounced degradation in accuracy (dropping to 74% and 68%, respectively) and require longer running times.

5.2 Optimal Graph Size for Training

5.2.1 Multi-Way Max-Cut

To determine the optimal graph size for training our GCN ~~on~~ for the multi-way max-cut problem, we conducted experiments with training sets containing graphs of varying sizes. Our main goal was to find the minimum number of large graphs needed in the training set so that the GCN can effectively scale to larger graphs while still achieving acceptable accuracy. Since training on larger graphs requires significantly more time, it is crucial to include only as many large graphs as necessary to keep the training time reasonable.

Experiment A: Forty-two unique training sets were created, each consisting of 400 graphs with sizes ranging from 200-500 nodes and degree ~~from~~ 6-12. A separate GCN model was training on these training sets ensuring that each neural network experienced a distinct training environment.

- 6 GCN models were trained with graphs of size between 200 and 500 ~~nodes~~
- 6 GCN models were trained with graphs of size between 200 and 500 with 2-6 graph ~~size~~ of 1000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 with 6-10 graph ~~size~~ of 1000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 with 10-14 graph ~~size~~ of 1000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 with 14-18 graph ~~size~~ of 1000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 with 18-22 graph ~~size~~ of 1000 nodes

To assess scalability and generalization, we tested the models on graphs with up to 8000 nodes. The testing set contained 20 graphs for each size range, with the mean average accuracy presented in the figures and tables below

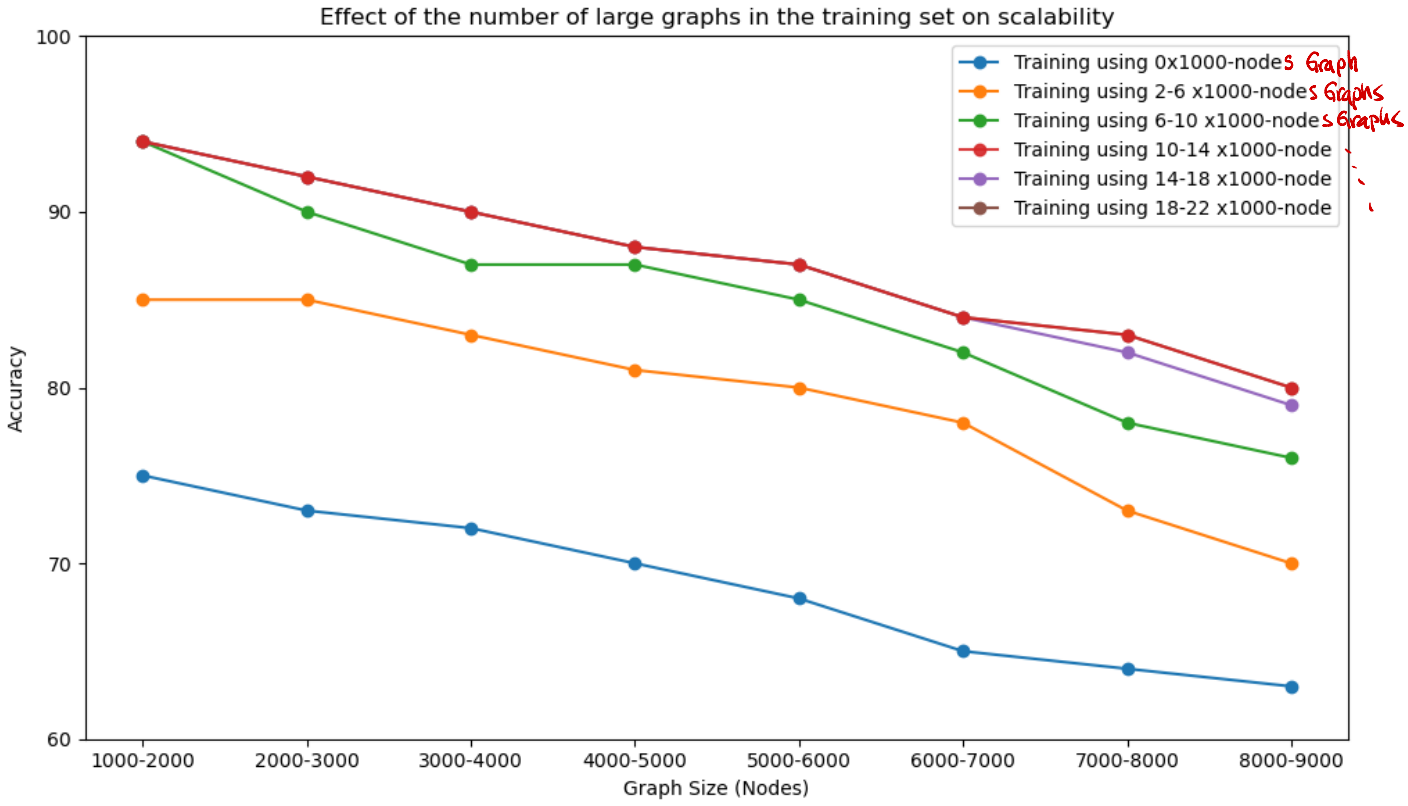


Figure 11: Impact of the number of 1000-node graphs in the training set on our GCN's scalability and performance. Some lines are not visible as they are hidden behind other lines.

Figure 11 shows the performance of our GCN for various training configurations with a few 1000-node graphs. Notably:

Table 1: Variances for the GCN models shown in Figure 11.

Training Setting	2000	3000	4000	5000	6000	7000	8000	9000
0 × 1000-node (blue)	2.5	2.7	2.9	3.1	3.2	3.3	3.4	3.5
2–6 × 6000-node	2.1	2.2	2.3	2.4	2.5	2.7	2.8	3.0
6–10 × 1000-node	1.4	1.6	1.7	1.8	2.0	2.1	2.2	2.3
10–14 × 1000-node	0.9	1.0	1.1	1.2	1.3	1.4	1.6	1.7
14–18 × 1000-node	0.9	1.0	1.0	1.1	1.2	1.3	1.4	1.5
18–22 × 1000-node	1.0	1.1	1.1	1.2	1.3	1.4	1.4	1.5

- Training with fewer than 10 large graphs (1000 nodes graphs) results in a significant drop in accuracy as the test graph sizes increase. For example, using only 2-6 larger graphs with 1000 nodes in the training set, the accuracy drops below 70% on graphs larger than 8000 nodes.
- Increasing the number of large graphs in the training set beyond 14 yields diminishing returns, as the accuracy improvement becomes marginal.
- For graphs larger than 8000 nodes, the performance of our GCN deteriorates rapidly, regardless of the training set size, emphasizing the challenge of scaling to graph sizes much larger than those used in the training set.
- ~~According to the figure, the optimal number of large graph (1000 node graph) needed are 10-14 large graphs in the training set.~~

Experiment B: ~~Another~~ Forty-two unique training sets were created, each consisting of 400 graphs with sizes ranging from 200-500 nodes and degree from 6-12. A separate GCN model was trained ~~on these~~ training sets.

- 6 GCN models were trained with graphs of size between 200 and 500.
- 6 GCN models were trained with graphs of size between 200 and 500 with 2-6 graph ~~size~~ of 6000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 with 6-10 graph ~~size~~ of 6000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 with 10-14 graph ~~size~~ of 6000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 with 14-18 graph ~~size~~ of 6000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 with 18-22 graph ~~size~~ of 6000 nodes

The results are shown in Figure 12 and variance is shown in Table 2.

Table 2: Variance values for Figure 12

Training Setting	1000	2000	3000	4000	5000	6000	7000	8000	9000
0 × 1000-node (blue)	2.8	2.5	2.9	3.3	3.1	2.9	2.7	3.0	3.2
2–6 × 6000-node (orange)	2.6	2.4	2.7	3.0	2.8	2.6	2.5	2.7	2.9
6–10 × 1000-node (green)	2.2	1.9	2.0	2.3	2.1	1.9	1.8	2.0	2.2
10–14 × 1000-node (red)	1.3	1.1	1.2	1.2	1.3	1.2	1.1	1.3	1.4
14–18 × 1000-node (brown)	1.3	1.1	1.1	1.2	1.2	1.2	1.1	1.3	1.4
18–22 × 1000-node (purple)	1.3	1.1	1.1	1.2	1.3	1.2	1.1	1.3	1.4

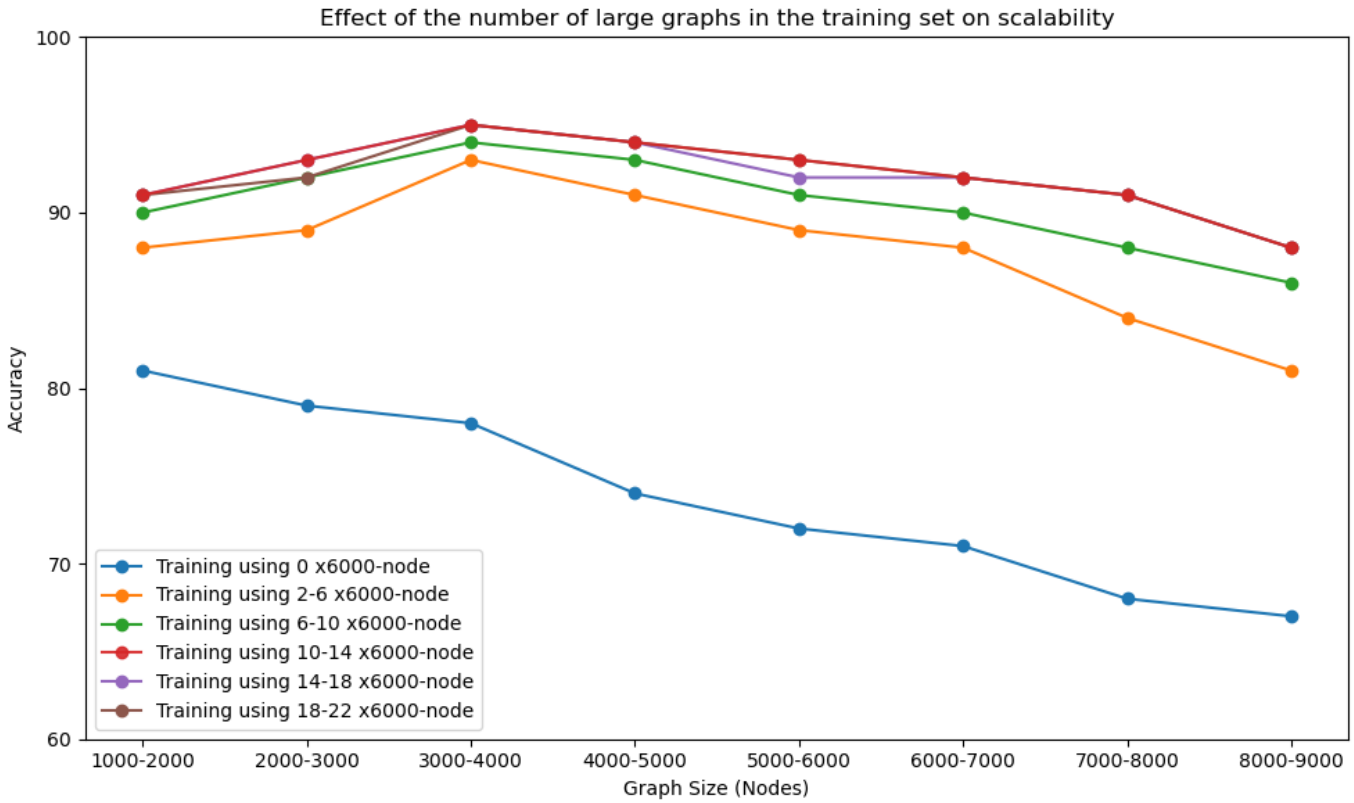


Figure 12: Impact of the number of 6000-node graphs in the training set on our GCN's scalability and performance.

From Figure 12, we observe the following:

- When training with 10-14 graphs of size 6000 nodes, the GCN achieves over 90% accuracy for test graphs up to 6000 nodes, demonstrating strong performance on graphs of similar sizes as ~~those~~ in the training set.
- Similar to the 1000-node case, at least 10-14 graphs with 6000 nodes are required in the training set to maintain scalability. However, there is performance degradation on graphs with more than 10,000 nodes.
- Including more than 14 ~~6000~~ graphs does not seem to improve scalability, ~~but the gain diminishes as the number of training set increases~~.

5.2.2 Balanced Multi-Way Max-Cut

In this section, we analyze the effect of the training set size and composition on our neural network's performance for the balanced multi-way max-cut problem. We performed similar experiments as for the multi-way max-cut problem.

Experiment A: Thirty unique training sets were created, each consisting of 400 graphs with sizes ranging from 200-500 nodes and degree from 6-12. A separate GCN model for balanced multi-way max-cut were trained on these training sets.

- 6 GCN models were trained with graphs of size between 200 and 500 nodes.
- 6 GCN models were trained with graphs of size between 200 and 500 nodes with 2-6 graph ~~size~~ of 1000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 nodes with 6-10 graph ~~size~~ of 1000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 nodes with 10-14 graph ~~size~~ of 1000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 nodes with 14-18 graph ~~size~~ of 1000 nodes
- 6 GCN models were trained with graphs of size between 200 and 500 nodes with 18-22 graph ~~size~~ of 1000 nodes

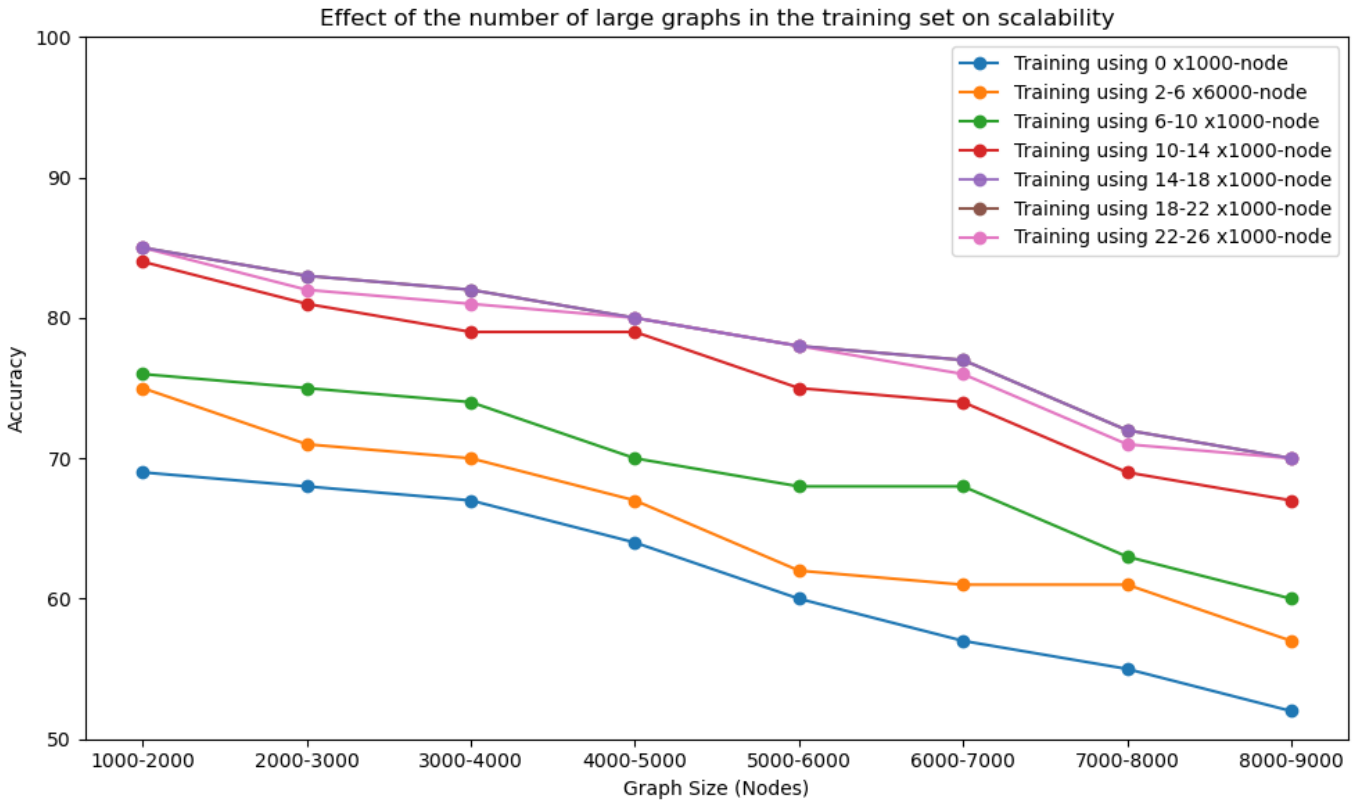


Figure 13: Impact of the number of 1000-node graphs in the training set on our GCN's scalability and performance.

Table 3: Variance for figure 13.

Training Setting	1000	2000	3000	4000	5000	6000	7000	8000
0 × 1000-node (blue)	2.6	2.4	2.2	2.1	1.9	2.1	2.3	2.4
2–6 × 6000-node (orange)	2.3	2.2	2.0	1.9	1.8	2.0	2.2	2.3
6–10 × 1000-node (green)	2.1	2.0	1.8	1.7	1.6	1.8	1.9	2.1
10–14 × 1000-node (red)	1.9	1.8	1.7	1.6	1.5	1.6	1.8	1.9
14–18 × 1000-node (purple)	1.2	1.3	1.4	1.4	1.4	1.5	1.6	1.7
18–22 × 1000-node (brown)	1.2	1.3	1.4	1.4	1.4	1.5	1.6	1.7
22–26 × 1000-node (pink)	1.2	1.3	1.4	1.4	1.4	1.5	1.6	1.7

From Figure 13, we observe the following:

- Training with fewer than 14 large graphs (1000 nodes graph) results in a significant drop in accuracy as the test graph sizes increase. For example, using only 2-6 larger graphs with 1000 nodes in the training set, the accuracy drops below 69% on graphs larger than 8000 nodes.
- Small increases in ~~Increasing~~ the number of large graphs in the training set beyond 18 yields diminishing returns, as the accuracy improvement becomes marginal.
- We observe using 14-18 graphs of size 1000 will yield ~~us~~ good and consistent results
- For graphs larger than 8000 nodes, the GCN performance of our GCN deteriorates rapidly, regardless of the training set size, emphasizing the challenge of scaling to graph sizes much larger than those used in the training set.

This is not shown by the figure.

General Observations Based on the experiments, the following observations can be made:

You should give training time for using only very large graphs or using only a few very large graphs.

1. **Balanced Multi-Way Max-cut:** At least 14-18 large graphs (1000 nodes) and a total of 800 graphs in the training set ~~are required to~~ scale effectively to graphs with up to 3 times the maximum graph size in the training set while maintaining ~~at least 80s~~ accuracy.
2. **Multi-Way Max-cut GCN:** At least 9-12 large graphs in a total of 600 graphs ~~are required~~ in the training set ~~to~~ scale effectively to graphs with up to 10 times the maximum graph size in the training set while maintaining ~~at least 80s~~ accuracy. We ~~can~~ also ~~take into account the fact~~ that Balanced Multi-way Max-cut GCN has no post-processing applied, while ~~the regular~~ Max-cut GCN has post-processing.

These results provide practical guidelines for designing training datasets for neural network-based combinatorial optimization, ~~particularly for balanced *multi*-way max-cut problems.~~

5.3 Effect of Graph Degree on Neural Network Training

5.3.1 Multi-way Max-cut

A key aspect of our investigation was to analyze the impact of graph degree on the efficacy of the neural network in solving the multi-way max-cut problem. To this end, we trained three separate GNN models on datasets with varying graph degrees.

- **GNN 1: High-Degree Graphs** — This model was trained on 300 graphs with high degrees, where the degree of each graph was set to $\lceil\sqrt{\text{nodes}}\rceil$.
- **GNN 2: Random-Degree Graphs** — This model was trained on 300 graphs with degrees randomly distributed between 6 and $\lceil\sqrt{\text{nodes}}\rceil$.
- **GNN 3: Low-Degree Graphs** - This model was trained on 300 graphs with relatively low degrees.

Performance on High-Degree Graphs When tested on high-degree graphs, Graph 1 and Graph 2 exhibited solution quality that was significantly closer to the integer solver's results compared to Graph 3. This is primarily because Graph 3, having been trained on low-degree graphs, struggled to generalize to the structure and complexity of high-degree graphs. The results, as illustrated in Fig. 3, highlight the limitations of training exclusively on low-degree graphs when applied to high-degree scenarios.

Performance on Low-Degree Graphs In contrast, when tested on low-degree graphs, all three models performed relatively well. However, it was observed that Graph 1, which was trained on high-degree graphs, underperformed slightly in isolation. Notably, when combined with a post-processing step, Graph 1's performance improved significantly, achieving results comparable to those of the other two models, as shown in Fig. 4. This indicates that post-processing can mitigate some of the limitations arising from training mismatches between graph degree distributions.

These observations emphasize the importance of aligning the training dataset's graph characteristics with the target application domain. Training on a diverse range of graph degrees or incorporating effective post-processing steps can enhance the neural network's ability to generalize across varying graph structures.

5.3.2 Balanced Multi-way Max-cut

The degree of graphs used in the training set plays a significant role in determining the performance of the neural network (NN) on balanced *multi*-way max-cut problems. To analyze this effect, we trained three versions of the NN: one trained on high-degree graphs, one on low-degree graphs, and another on graphs with a mix of random degrees. The trained models were then tested on graphs with both high and low degrees, and the results are presented in Figures 16 and 17.

Performance on High-Degree Graphs Figure 16 shows the performance of the three neural networks compared to the Integer Solver (CPLEX) when tested on high-degree graphs:

- The NN trained on high-degree graphs performed the best, maintaining approximately 83% accuracy for smaller graphs (1000-2000 nodes) and dropping to 73% for larger graphs (5000-6000 nodes). This highlights its strong generalization to high-degree graphs.
- The NN trained on random-degree graphs performed slightly worse, achieving 81% accuracy for smaller graphs but dropping to 71% for larger graphs. The mixed training set provided moderate generalization but was outperformed by the high-degree-trained NN.
- The NN trained on low-degree graphs showed significant performance degradation, achieving only 73% accuracy for smaller graphs and dropping to 60% for larger graphs. This indicates its limited ability to generalize to high-degree test graphs.

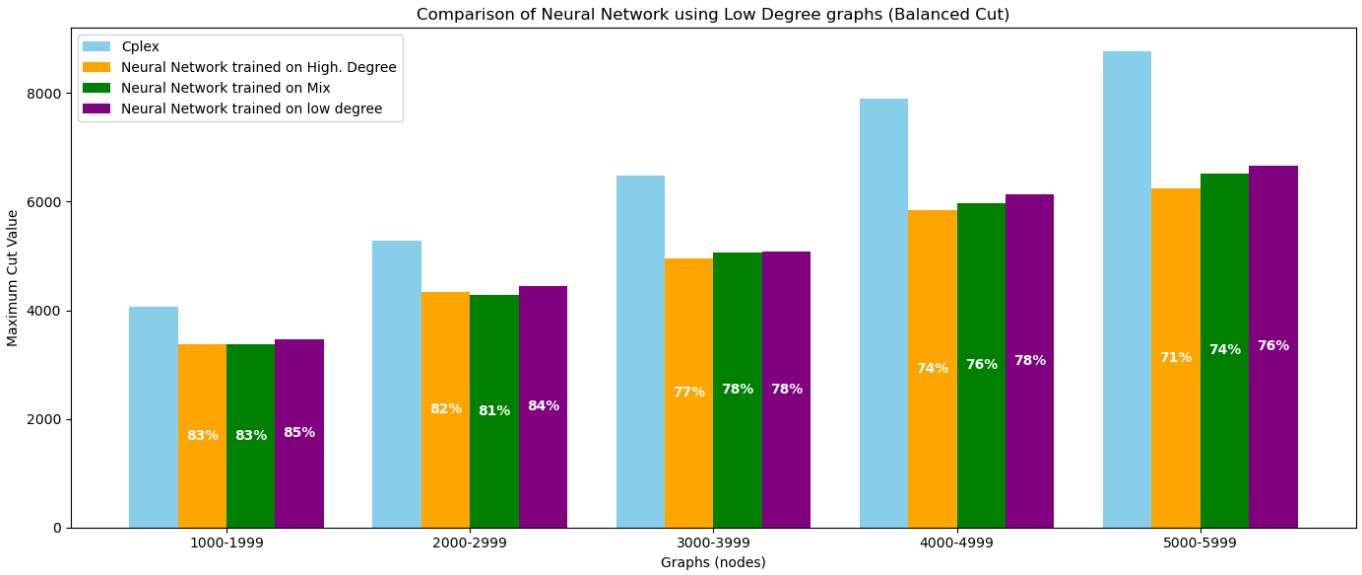


Figure 16: Performance comparison of neural networks trained on high-degree, low-degree, and random-degree graphs when tested on high-degree graphs. The NN trained on high-degree graphs shows superior scalability and accuracy.

Performance on Low-Degree Graphs Figure 17 illustrates the performance of the same neural networks on low-degree graphs:

- The NN trained on low-degree graphs achieved the best results, maintaining 85% accuracy for smaller graphs (1000-2000 nodes) and approximately 76% accuracy for larger graphs (5000-6000 nodes).
- The NN trained on random-degree graphs performed comparably to the low-degree-trained NN, with only minor differences in accuracy. This highlights the robustness of a mixed training set for low-degree graphs.
- The NN trained on high-degree graphs performed the worst, achieving 83% accuracy for smaller graphs but dropping to 71% for larger graphs, indicating limited generalization to low-degree test graphs.

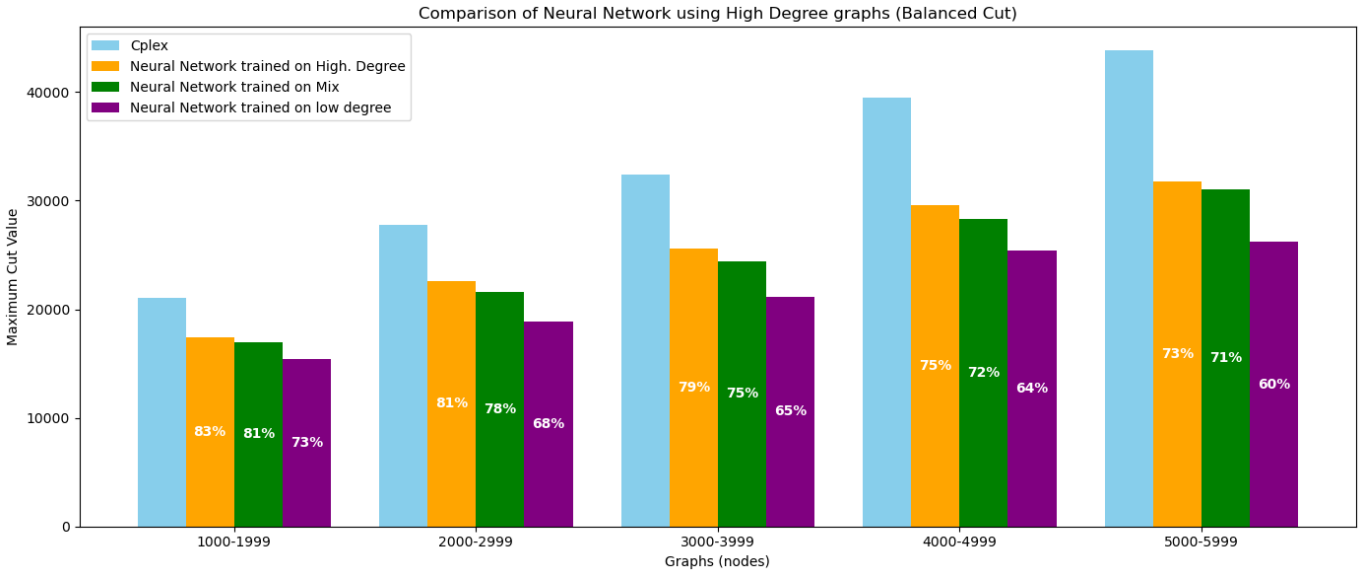


Figure 17: Performance comparison of neural networks trained on high-degree, low-degree, and random-degree graphs when tested on low-degree graphs. The NN trained on low-degree graphs shows superior accuracy and scalability.

Key Observations From the results, we observe:

1. **Degree-Specific Training:** Neural networks trained on high-degree graphs generalize well to high-degree test graphs but struggle with low-degree graphs. Similarly, NNs trained on low-degree graphs excel on low-degree test graphs but perform poorly on high-degree graphs.
2. **Mixed-Degree Training:** Training on graphs with random degrees provides moderate performance across both high-degree and low-degree test graphs, indicating that a diverse training set can improve generalization across graph types.
3. **Scalability Challenges:** All neural networks exhibit performance degradation as the graph size increases, particularly for test graphs far larger than those seen during training.

5.3.3 Impact on Neural Network Accuracy

The post-processing algorithm enhances the neural network's accuracy by refining initial predictions and ensuring better alignment with the objective function. Our experiments show that post-processing consistently improves solution quality by 8-10%, as it effectively searches for higher-quality partitions within the space of potential solutions. Additionally, the algorithm scales efficiently with the size of the graph, as each iteration operates independently, allowing for parallelization if required. This makes it a practical and impactful addition to the neural network pipeline for solving combinatorial optimization problems.

The experiments demonstrate the importance of aligning the training set's degree distribution with the target graph properties to achieve optimal performance. While degree-specific training yields the best results for corresponding test graphs, mixed-degree training provides a balanced trade-off for applications requiring generalization across varying graph types. These findings underscore the need for carefully curated training datasets to address the diverse requirements of balanced *multi*-way max-cut problems.

6 Conclusion

The results of this study highlight the effectiveness of the proposed GNN-based framework for solving NP-hard combinatorial optimization problems, specifically the multi-way max-cut and balanced multi-way max-cut. By leveraging the scalability and efficiency of Graph Neural Networks (GNNs) in an unsupervised learning paradigm, our approach

achieves good accuracy solutions while significantly outperforming traditional integer solvers in terms of computational speed. Despite the additional constraints introduced in the balanced multi-way max-cut problem, our method demonstrates competitive performance while maintaining efficiency. The ability of our model to generalize to larger graphs, even when trained on significantly smaller graphs, underscores its robustness and potential for real-world applications. This suggests that with further fine-tuning and improved training strategies, we can enhance both accuracy and scalability, making neural network-based combinatorial optimization a viable alternative to conventional methods. One of the most promising aspects of this work is the potential for reducing training time while still achieving high-quality solutions. By refining training methodologies—such as optimizing hyperparameters, utilizing transfer learning, and employing more structured datasets—we can further improve the model's ability to scale across graph sizes. The observation that training on small graphs can yield strong performance on significantly larger graphs highlights the efficiency gains possible with this approach, potentially reducing computational costs and making large-scale optimization problems more tractable. The impact of this research extends beyond graph partitioning problems. The demonstrated efficiency of GCN-based approaches for solving NP-hard problems opens avenues for broader applications in network optimization, logistics, circuit design, and beyond. Future work will focus on refining the loss function, improving post-processing heuristics, and further enhancing generalization to diverse problem instances. With continued advancements, neural network-based combinatorial optimization has the potential to bridge the gap between traditional solvers and real-time, scalable decision-making for large-scale problems.

References

[1] M. J. A. Schuetz, J. Brubaker, and H. G. Katzgraber, "Combinatorial Optimization with Physics-Inspired Graph Neural Networks", arXiv:2107.01188, 2022. Available: https://arxiv.org/pdf/2107.01188v2.pdf.

[2] H. Sun, E. K. Guha, and H. Dai, "Annealed Training for Combinatorial Optimization on Graphs", OPT2022: 14th Annual Workshop on Optimization for Machine Learning arXiv:207.11542, 2022. Available: https://arxiv.org/pdf/207.11542v1.pdf.

[3] Z. Li, Q. Chen, and V. Koltun, "Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search," NIPS, 2018. Available: https://neurips.cc/Conferences/2018/Schedule?showEvent=11077.

[4] N. Karalias and A. Loukas, "Erdős Goes Neural: An Unsupervised Learning Framework for Combinatorial Optimization on Graphs," NIPS, 2020. Available: https://proceedings.nips.cc/paper$_files/paper/2020/file/49f85a9ed090b20c8bed85a5923c669f-Paper.pdf$.

[5] H. Dai, E. B. Khalil, Y. Zhang, and B. Dilkina, "Learning Combinatorial Optimization Algorithms over Graphs.", NIPS, 2017, Available: https://proceedings.neurips.cc/paper$_files/paper/2017/file/d9896106ca98d3d05b8cbdf4fd8b13a1-Paper.pdf$.

[6] W. Yao, A. S. Bandeira, and S. Villar, "Experimental Performance of Graph Neural Networks on Random Instances of Max-Cut.", Proceedings Volume 11138, Wavelets and Sparsity XVIII, 2019

[7] G. Kochenberger, J.-K. Hao, F. Glover, M. Lewis, Z. Lu, H. Wang, and Y. Wang, "The Unconstrained Binary Quadratic Programming Problem: A Survey," *Journal of Combinatorial Optimization*.

[8] B. Korte and J. Vygen, *Combinatorial Optimization*, vol. 2, Springer, New York, 2012.

[9] A. Barzegar, C. Pattison, W. Wang, and H. G. Katzgraber, "Optimization of Population Annealing Monte Carlo for Large-Scale Spin-Glass Simulations," *Phys. Rev. E*, vol. 98, p. 053308, 2018.

[10] G. A. Kochenberger, J.-K. Hao, Z. Lu, H. Wang, and F. Glover, "Solving Large-Scale Max Cut Problems via Tabu Search," *Journal of Heuristics*, vol. 19, p. 565, 2013.

[11] K. Li and J. Malik, "Learning to Optimize," arXiv preprint arXiv:1606.01885, 2016.

[12] H. Djidjev, P. Chapuis, G. Hahn, and G. Rizk, "Efficient Combinatorial Optimization Using Quantum Annealing."

[13] V. Air, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, et al., "Solving Mixed Integer Programs Using Neural Networks," arXiv preprint arXiv:2012.13349, 2020.

[14] H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning," in *31st Conference on Neural Information Processing Systems (NIPS)*.

[15] R. Wang, J. Yan, and X. Yang, "Combinatorial Learning of Robust Deep Graph Matching: An Embedding-Based Approach," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[16] D. Selsam and N. Bjørner, "Guiding High-Performance SAT Solvers with Unsat-Core Predictions," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 336–353, Springer, 2019.

[17] W. Yao, A. S. Bandeira, and S. Villar, "Experimental Performance of Graph Neural Networks on Random Instances of Max-Cut," 2019.

[18] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer Networks," in *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

[19] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna, "A Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks," *Stat*, vol. 1050, p. 22, 2017.

[20] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang, "Graph Edit Distance Computation via Graph Neural Networks," arXiv preprint arXiv:1808.05689, 2018.

[21] H. Lemos, M. Prates, P. Avelar, and L. Lamb, "Graph Colouring Meets Deep Learning: Effective Graph Neural Network Models for Combinatorial Problems," 2019.

[22] Z. Li, Q. Chen, and V. Koltun, "Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search," in *Advances in Neural Information Processing Systems*, pp. 539–548, 2018.

[23] C. K. Joshi, T. Laurent, and X. Bresson, "An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem," arXiv preprint arXiv:1906.01227, 2019.

[24] N. Karalias and A. Loukas, "Erdős Goes Neural: An Unsupervised Learning Framework for Combinatorial Optimization on Graphs," arXiv preprint arXiv:2006.10643, 2020.

[25] G. Yehuda, M. Gabel, and A. Schuster, "It's Not What Machines Can Learn, It's What We Cannot Teach," arXiv preprint arXiv:2002.09398, 2020.

[26] D. Selsam and N. Bjørner, "Guiding High-Performance SAT Solvers with Unsat-Core Predictions," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 336–353, Springer, 2019.

[27] F. M. Bianchi, D. Grattarola, and C. Alippi, "MinCut Pooling in Graph Neural Networks," 2019.

[28] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, "Exact Combinatorial Optimization with Graph Convolutional Neural Networks," arXiv preprint arXiv:1906.01629, 2019.

[29] P.-W. Wang, P. L. Donti, B. Wilder, and Z. Kolter, "SatNet: Bridging Deep Learning and Logical Reasoning Using a Differentiable Satisfiability Solver," arXiv preprint arXiv:1905.12149, 2019.

[30] M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rolínek, "Differentiation of Blackbox Combinatorial Solvers," arXiv preprint arXiv:1912.02175, 2019.

[31] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural Combinatorial Optimization with Reinforcement Learning," arXiv preprint arXiv:1611.09940, 2017.

[32] J. Kotary, F. Fioretto, P. Van Hentenryck, and B. Wilder, "End-to-End Constrained Optimization Learning: A Survey," arXiv preprint arXiv:2103.16378, 2021.

[33] W. Kool, H. van Hoof, and M. Welling, "Attention, Learn to Solve Routing Problems!" arXiv preprint arXiv:1803.08475, 2019.

[34] Q. Ma, S. Ge, D. He, D. Thaker, and I. Drori, "Combinatorial Optimization by Graph Pointer Networks and Hierarchical Reinforcement Learning," arXiv preprint arXiv:1911.04936, 2019.

[35] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning Combinatorial Optimization Algorithms Over Graphs," arXiv preprint arXiv:1704.01665, 2018.

[36] Y. Bai, D. Xu, A. Wang, K. Gu, X. Wu, A. Marinovic, C. Ro, Y. Sun, and W. Wang, "Fast Detection of Maximum Common Subgraph via Deep Q-Learning," arXiv preprint arXiv:2002.03129, 2020.

[37] L. Gao, M. Chen, Q. Chen, G. Luo, N. Zhu, and Z. Liu, "Learn to Design the Heuristics for Vehicle Routing Problem," arXiv preprint arXiv:2002.08539, 2020.

[38] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement Learning for Solving the Vehicle Routing Problem," in *Advances in Neural Information Processing Systems*, pp. 9839–9849, 2018.

[39] J. Q. James, W. Yu, and J. Gu, "Online Vehicle Routing with Neural Combinatorial Optimization and Deep Reinforcement Learning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 10, pp. 3806–3817, 2019.

[40] S. Thrun and A. Schwartz, "Issues in Using Function Approximation for Reinforcement Learning," in *Proceedings of the 1993 Connectionist Models Summer School*, Hillsdale, NJ, Lawrence Erlbaum, 1993.

[41] E. Nikishin, P. Izmailov, B. Athiwaratkun, D. Podoprikhin, T. Garipov, P. Shvechikov, D. Vetrov, and A. G. Wilson, "Improving Stability in Deep Reinforcement Learning with Weight Averaging," in *Uncertainty in Artificial Intelligence Workshop on Uncertainty in Deep Learning*, vol. 5, 2018.

[42] A. Irpan, "Deep Reinforcement Learning Doesn't Work Yet," 2018. Available: https://www.alexirpan.com/2018/02/14/rl-hard.html.

[43] E. Yolcu and B. Poczos, "Learning Local Search Heuristics for Boolean Satisfiability," in *Advances in Neural Information Processing Systems*, pp. 7990–8001, 2019.

[44] X. Chen and Y. Tian, "Learning to Perform Local Rewriting for Combinatorial Optimization," in *Advances in Neural Information Processing Systems*, pp. 6278–6289, 2019.

[45] S. Even, A. Itai, and A. Shamir, "On the Complexity of Timetable and Multicommodity Flow Problems," *SIAM Journal on Computing*, 1976.

[46] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, 1970.

[47] M. E. J. Newman, "Modularity and Community Structure in Networks," *Proceedings of the National Academy of Sciences*, 2006.

[48] J. Shi and J. Malik, "Normalized Cuts and Image Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2000.

[49] C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey," *Integration, the VLSI Journal*, 1995.

[50] F. Barahona, M. Grötschel, M. Junger, and G. Reinelt, "An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design," *Operations Research*, vol. 36, pp. 493–513, 1988.

[51] Y. J. Xi and Y. Z. Dang, "The Method to Analyze the Robustness of Knowledge Network Based on the Weighted Supernetwork Model and Its Application," *Systems Engineering Theory and Practice*, vol. 27, pp. 134–140, 2007.

[52] S. Dreiseitl and L. Ohno-Machado, "Logistic Regression and Artificial Neural Network Classification Models: A Methodology Review," *Journal of Biomedical Informatics*, vol. 35, pp. 352–359, 2002.

[53] F. M. Bianchi, D. Grattarola, and C. Alippi, "Mincut Pooling in Graph Neural Networks," 2019.

[54] K. A. Smith, "Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research," *INFORMS Journal on Computing*, vol. 11, no. 1, pp. 15–34, 1999.

[55] J. J. Hopfield and D. W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics*, vol. 52, no. 3, pp. 141–152, 1985.

[56] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[57] https://www.ibm.com/products/ilog-cplex-optimization-studio

[58] "Improved Approximation Algorithms for MAX k-CUT and MAX BISECTION" by Goemans and Williamson (2004)

[59] "The Max k-Cut Problem on Classical and Quantum Solvers" by Alghassi et al. (2021)

7 Glossary

7.0.1 Neighbor Set $\mathcal{N}(u)$

For each node u, its neighbor set $\mathcal{N}(u)$ consists of all nodes directly connected to it via an edge. These neighbors provide contextual information that is aggregated at each GNN layer. The size and structure of $\mathcal{N}(u)$ significantly impact the quality of learned node embeddings, with denser connections leading to richer representations.

7.0.2 Node Representation $h_u^{(l)}$

At each layer l, every node u is associated with a hidden representation $h_u^{(l)}$, which encodes the node's current state based on its own features and those of its neighbors. Initially, $h_u^{(0)}$ is set to the node's raw features. As information propagates through the layers, the representation $h_u^{(l)}$ becomes more expressive, capturing higher-order dependencies in the graph.

7.0.3 Input Features

Input Features are set of features that are specific to the input graph that is being sent to the GCN. In our case, input feature is a n by n matrix, there n is number of nodes and value at i,j indicates the edge weight between the nodes.

7.0.4 Aggregation: Information from Local Neighborhoods

A core component of GNNs is the aggregation mechanism which allows each node to incorporate information from its neighbors to update its own representation.

7.0.5 Learnable Weight Matrix $W^{(l)}$

Each layer of a GNN has an associated learnable weight matrix $W^{(l)}$, which transforms node embeddings into a new feature space. The weights are optimized through gradient-based learning (e.g., stochastic gradient descent) to enhance the ability of the model to capture meaningful graph structures.% [**?**].

7.0.6 Final Node Representation

After L layers, the final node representations $h_u^{(L)}$ encode structural and feature information from a node's local and extended neighborhood. These representations are then used for graph-based learning tasks, often through additional layers such as fully connected layers or pooling operations.% [**?**].

7.0.7 Update Rule

The update rule defines how a node's representation is updated using the aggregated messages from its neighbors. The update process typically consists of:

1. Message Passing: Each node collects feature information from its neighbors.
2. Aggregation: Neighboring node features are combined using a permutation-invariant function such as sum, mean, or max pooling.
3. Transformation: A learnable function, often a linear transformation with weights $W^{(l)}$, is applied to the aggregated features.
4. Non-Linearity: A non-linear activation function σ (e.g., ReLU) is applied to enhance expressiveness.

These steps allow the GNN to gradually refine each node's representation by incorporating information from larger regions of the graph over multiple layers.

7.0.8 Message Function ϕ

The message function ϕ determines how information is transferred between connected nodes. Different GNN variants use different message functions:

- Simple Summation: $\phi(h_u, h_v) = h_v$ (Graph Convolutional Networks.
- Attention Mechanism: Assigns different weights to neighbors (Graph Attention Networks).
- Edge-Weighted Propagation: Incorporates edge features explicitly (GraphSAGE).

7.0.9 Multi-Hop Neighbors

In a graph $G = (V, E)$, the *multi-hop neighbors* of a node u refer to the set of nodes that are reachable from u within multiple hops along the edges of the graph. Formally, the k-hop neighborhood of a node u, denoted as $\mathcal{N}^{(k)}(u)$, consists of all nodes that can be reached from u in at most k steps:

$$\mathcal{N}^{(k)}(u) = \{v \in V \mid \text{shortest path}(u, v) \leq k\}.$$

In Graph Convolutional Networks (GCNs), stacking multiple layers allows each node to iteratively aggregate information from its multi-hop neighbors, thereby capturing both local and global graph structures. This mechanism is crucial for learning richer node representations in tasks such as node classification, link prediction, and combinatorial optimization.

7.0.10 Euclidean data

Data that is represented in a continuous, flat, and finite-dimensional space. Examples include images (grids of pixels), time-series data (sequences), and structured tabular data, where the notion of distance between points is defined using Euclidean geometry.