University of Bergen
Department of Informatics

Solving Maximum Weighted Matching problem using Graph Neural Networks

Author: Nikita Zaicev

Supervisor: Fredrik Manne

UNIVERSITETET I BERGEN

Det matematisk-naturvitenskapelige fakultet

June, 2024

Abstract

In this work we train a Graph Neural Network model to solve the Maximum Weighted Matching problem on graphs using supervised learning. Unfortunately, the final results were below the set expectations. The model seemed to be slightly worse than a standard greedy algorithm, mainly because the greedy algorithm on average performed very well compared to the optimal solution. However, the neural network did show potential at solving the problem for more narrow graph datasets and graphs that were particularly difficult for the greedy algorithm. The results do not imply that graph neural networks in general cannot beat greedy algorithms, but rather suggest that different approaches or further improvement are needed.

Acknowledgements

I want to thank Fredrik Manne, Kenneth Langedal and Johannes Langguth for helping me with this work.

Nikita Zaicev
Saturday 1st June, 2024

Contents

1 Introduction **1**
1.1 Goal . . . 1
1.2 Motivation . . . 2
1.3 Project structure . . . 3
1.3.1 Git . . . 4
1.3.2 Writing . . . 4

2 Background **5**
2.1 Neural Networks and Machine Learning . . . 5
2.2 Graphs . . . 9
2.3 Graph Neural Networks . . . 11
2.4 Implementation, training and hyperparameters . . . 14
2.4.1 Implementation . . . 14
2.4.2 Training . . . 15
2.4.3 Hyperparameters . . . 16
2.5 Maximum Weighted Matching and Combinatorial Optimization . . . 18

3 Research Methodology and Data **20**
3.1 Challenges . . . 20
3.2 Main Idea . . . 21
3.3 Data . . . 23
3.4 Architecture . . . 24
3.4.1 Model pipeline . . . 26
3.4.2 Line Graph Approach . . . 26
3.4.3 Edge Classification Approach . . . 28
3.4.4 Data preprocessing . . . 29
3.5 Result Validation . . . 31
3.5.1 Sanity checks . . . 32
3.6 Expected Results . . . 32

3.6.1 Other observations . . . 32
3.6.2 Accuracy and total weight . . . 33
3.6.3 Time . . . 33
3.6.4 Remainder or model portion . . . 33

4 Results **34**
4.1 Progress . . . 34
4.1.1 Experimenting environment . . . 34
4.1.2 Simple line graph model . . . 34
4.1.3 Model improvements and data augmentation . . . 36
4.2 Edge prediction model . . . 39
4.2.1 Reduction . . . 44
4.2.2 Matching threshold . . . 45
4.3 Final results . . . 47
4.3.1 Weakness of the greedy algorithm . . . 49

5 Conclusion **51**
5.1 Future work . . . 52

Glossary **53**

List of Acronyms and Abbreviations **54**

Bibliography **55**

A Code examples **59**

List of Figures

2.1 Simple neural network example 7
2.2 Classification example 8
2.3 The difference between a directed and an undirected graph 9
2.4 Original graph (left) and its line graph (right) 10
2.5 GNN example 12
2.6 Message passing example 13
2.7 GCN example 14

3.1 Original graph 28
3.2 Line graph 28
3.3 Line graph convertion example 28

4.1 MNIST trained model. Average performance on 100 MNIST graphs . . . 38
4.2 MNIST performance comparison 40
4.3 MNIST trained model performance on cage10 graph 41
4.4 MNIST vs CUSTOM dataset training model performance on cage10 . . . 42
4.5 CUSTOM trained model performance on cage10 graph 42
4.6 MNIST vs CUSTOM dataset training model performance on MNIST graphs 43
4.7 Reduction example 44
4.8 Model threshold test on cage8 graph 46
4.9 Final model performance 48
4.10 Good case for GNN 49
4.11 Greedy disadvantaged graphs 50

List of Tables

3.1 Names and sizes of graphs used for testing 24

4.1 Performance improvement from increasing the depth of the mode 36
4.2 Performance improvement from increasing the width of the model 36
4.3 Aggregation function performance 39

Listings

2.1 Creating GCN with Pytorch 14
2.2 Training GCN with Pytorch 16
3.1 Pytorch GCN for the line graph 25
3.2 Skip connection example 25
3.3 Pytorch GCN for the edge classification 29

Chapter 1

Introduction

This chapter is an introduction that explains the goals and motivation for this work.

1.1 Goal

Classification is one of the most fundamental concepts in machine learning. It involves categorizing data into distinct subclasses based on their features. For instance, consider a dataset containing objects with an annotated feature indicating their color, and three classes: apples, oranges. Using the color feature, we can classify the objects into two distinct categories. Different variations of such classification problems occur across many scientific fields and machine learning has been a potent tool for solve such problems. Machine learning includes a variety of statistical methods for solving classification problems. One such method is neural networks.

A popular example of neural network application is recognition of handwritten digits [11], where one can think of each pixel in an image as a feature that a neural network can use to classify the digit on the image. Examples of classification problems that can be solved using neural networks can be found across a variety of fields including: feature based classification of Iris plants [25], image recognition for classification of cholera and malaria pathogens [26] and credit card fraud detection [8]. For such classification problems, input data often has an expected structure. Input features such as, for example, image pixels are often expected to be of a certain size or are preprocessed accordingly. The neural networks have, however, shown promising results when dealing with unstructured

data such as input of varying length in the form of text or video. Some well known examples are: text generation for the popular chatbot "ChatGPT" [22], photorealistic text-to-image synthesis [16] or self-driving cars [3]. Another example of unstructured data is a graph. As an analogy, a picture can be represented as a grid graph where each pixel is connected to the pixels it is surrounded by. Such a graph would have a set number of neighbours and a set width and height. A simple graph, however, does not have such limitations and each element can have a varying number of neighbours as well as a variable width and height.

Combinatorial Optimization (CO) is a field of study that often deals with algorithmic problems related to graphs and has been a growing, yet challenging research topic in machine learning for the last few years. CO includes graph problems such as Maximal Independent Set (MIS) and Maximum Weighted Matching (MWM), with MWM being the main topic of this work. In simple terms, MWM problem requires one to match together pairs of objects such that the total weight is maximized, given a list of all the possible pairing combinations with weights assigned to them. As a simplified example, one can view the problem in the context of task scheduling. For instance, consider a company that wants to assign workers to a set of tasks and maximize the overall performance. Each worker has different skills and is expected to perform better on some projects than others. How well a person fits for a project can be expressed by some metric and workers can only work on one project at a time. In other words, the company would want to match employees to the projects such that the total performance of the workers is as high as possible, resembling the MWM problem. Variations of such scheduling problems arise in various fields, including processor scheduling [7] and donor compatibility matching [23]. Additionaly, MWM solvers are used in larger algorithms. To solve algorithmic problems related to graphs, a specific subclass of neural networks has been designed called Graph Neural Network (GNN)s. However, there are still problems that have not yet been solved efficiently with GNNs.

The main goal for this project is to: "Find out whether a GNN can outperform approximation algorithms such as greedy in solving MWM problem".

1.2 Motivation

The previous section describes "what" this work is about. Now let's briefly discuss "why" do this in the first place. The idea is rather simple. There are two types of algorithms

in general: exact and approximate. Exact algorithms guarantee that the found solution is optimal and correct, meaning that there cannot be a better solution, although several equally good solutions might exist. Approximation algorithms are heuristic techniques that do not aim for perfect optimalization, but rather focus on practical approaches that give sufficient results for the required task. These algorithms do not guarantee the optimal solution, but they provide a close approximation with a defined minimum level of precision. Similarly, machine learning and neural networks also fall into the category of heuristics.

The exact algorithms for MWM already exist, but the downside of such algorithms is their speed. The fastest algorithm is due to Edmonds and runs in polynomial time $O(|V|^2|E|)$ [29], where $|V|$ denotes the number of nodes in a graph and $|E|$ denotes the number of edges. There are several approximation algorithms for MWM. One of the most simple ones is a weight based greedy approximation algorithm that matches pairs based on the their weights sorted in descending order. Such algorithm runs in time $O(|E|log(|E|))$ with the main factor being the sorting time. The slower nature of exact algorithms is the reason why approximation algorithms are used in some cases. The hope is that a GNN model can potentially squeeze in between the approximation and exact algorithms in terms of both time and accuracy.

In short, the motivation is: "There can be a machine learning based algorithm that gives better results than existing approximation algorithms"

1.3 Project structure

The rest of this document has the following structure:

1. Background:

   This Chapter briefly outlines the history, applications and impact of machine learning and goes through core concepts and ideas behind machine learning, neural networks and graph neural networks. Relevant information about CO and algorithms is included here.

2. Methodology and Data:

   This Chapter focuses on how the research was done and also explains core concepts of Neural Networks and the general procedure of training a Neural Network. Then

the specifics of this case are discussed together with the main challenges. The progress made step by step and the reasoning behind changes and choices made along the way are shown. The chapter also analyzes the data used for training the model and evaluating results, along with justification for the chosen data.

3. Results:

   This Chapter focuses on temporary results produced during the research and explains how these results affected further decisions. Finally, the chapter is concluded by analyzing the final results and comparing them with expectations.

4. Conclusion:

   Here the conclusion for this project is drawn regarding whether the results gave any meaningful insight and what future work can be done for improvements.

1.3.1 Git

The code used in this project can be accesssed at: `https://github.com/nikitazaicev/Master`. The information about the used datasets is given in the Data analysis chapter

1.3.2 Writing

ChatGPT was used to improve wording and grammar for some small parts of the text in this report.

Chapter 2

Background

This Chapter focuses on the history, applications and impact of machine learning and goes through core concepts and ideas behind neural networks and graph neural networks. Relevant information about CO, graphs, exact and approximation algorithms is also included here.

2.1 Neural Networks and Machine Learning

Machine learning in computer science is a broad term for all the models and algorithms that use statistical methods to learn patterns and make predictions. Some of the earliest algorithms were invented in the 1940s [12], but the viability was limited by the lack of data and computational power. One of the first works that began using GPUs to speed up the training process were Raina et al. [20] in 2009. Both GPU and machine learning technologies have been improving ever since. The use of neural networks has found a place in almost every field.

Neural network is a statistical method in machine learning inspired by the way brains work. The areas in which this method is applied include classification tasks that aim to identify certain objects based on their features, such as Iris flower classification [25] presented by Swain et al. A family of Iris flowers can be split into three categories based on the features, such as the shapes of petals and sepals. Iris flower classification is a good and simple example of a classification problem where the input is the lengths and widths of petals and sepals, and the task of the neural network is to put every flower

in the correct category based on these input features. Other complex problems such as image recognition of pathogens [26] have also been solved by using neural networks. Besides recognition and classification, neural networks have also found applications in the synthesis of data, such as text generation with ChatGPT [22] and image generation [16]. From around 2017, the Graph Neural Network (GNN) subclass of neural networks started to gain more traction [9], [28], [15]. The GNNs are specialized for working with graph data as the name suggests and are the main topic of this work, but before describing the GNNs in more detail, let us revisit the basics of machine learning and neural networks.

Machine learning can be split into supervised, unsupervised and reinforcement learning.

In the case of supervised learning, the machine learning model is given training data with the correct answers. The model can then learn from this by trial and error and then apply gained knowledge to the unseen data.

Supervised learning requires a dataset with precomputed answers. It is common to split the dataset into three parts: training, validation and test. Training is used to train the model, while validation is used to evaluate several models trained on the same training dataset, but with different parameters for models learning process. The test dataset is the final data set used to evaluate the final model, since the model was chosen based on the validation dataset and needs to be tested on unseen data to avoid bias. This approach relies on feeding the model large amounts of data in order to cover as much variety in data as possible. The training process can take days, but on the positive side, after the model is done with training, the running time of making a prediction can be as low as $O(n)$ in the case of simple neural networks, although it depends on the architecture.

In the case of unsupervised learning, the model lacks the correct answers and has to use other means to calculate the error and adjust weights. Autoencoders are an example of such networks and rely on two separate neural networks, where one part attempts to encode data and the other to decode it and give feedback based on how close the decoded data is to the original.

Reinforcement learning is less relevant, since it is used in cases with no training data. And in this work we have data in the form of graphs. Therefore, in this thesis, we are using supervised learning, where a neural network is trained on a dataset of graphs with optimal solutions found using an exact algorithm.

Neural networks are a subclass of machine learning algorithms that are inspired by the human brain. They are, for the most part, made of different layers that consist of

“neurons”. Layers can be grouped into three types: an input layer that reads the given data, one or more middle layers that process the data and an output layer that gives the final answer. In a simple neural network, each neuron in a layer is usually connected to all the neurons of the next layer, although it often depends on the architecture.

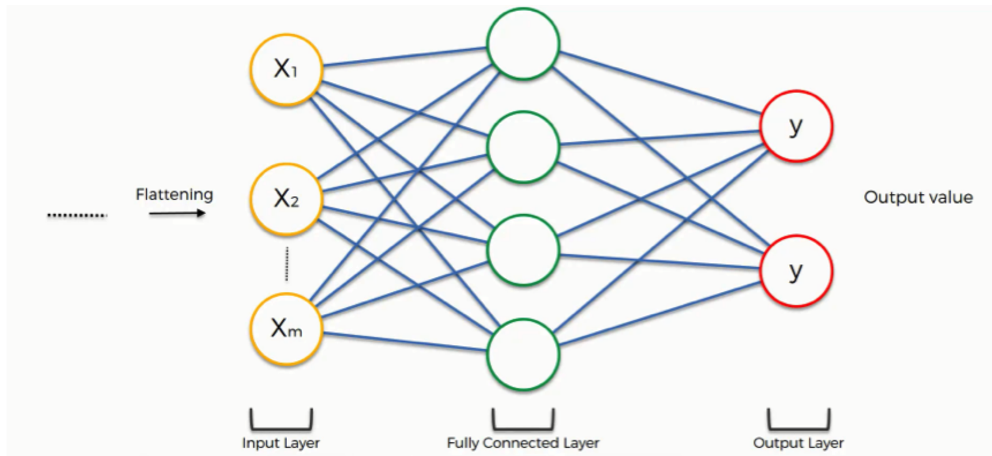

Figure 2.1: Simple neural network example
Source: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-4-full-connection

Figure 2.1 visualizes how a simple fully connected neural network might look. The flattening refers to formatting the input features as a one-dimensional array to match the input layer size. This type of neural network is called fully connected, since each neuron is connected to every neuron in the next layer. The output layer represents the final binary classification step, where the two neurons stand for each of the two total categories this network is designed to identify.

Neuron connections have assigned weights. When the neural network model receives input, it is passed through the layers and gets multiplied by the weights. Assume an input neuron with value $x1$ is passed to a neuron in the next layer. The value of the neuron in the next layer will then be $w1(x1) + b1$, where $w1$ is the weight between the nodes and $b1$ is the bias term used to improve models flexibility. One might have noticed that the formula for passing data from one neurn to the other is a simple linear function. After the passing is done and the output layer is reached, the result in the output layer is some altered numerical representation of the input that can, depending on the goal, be interpreted as some prediction. For example, by applying a Softmax function on the output of the model, the result of each neuron would be between 0 and 1, and could be treated as the probabilities. This is a common way to do classification tasks.

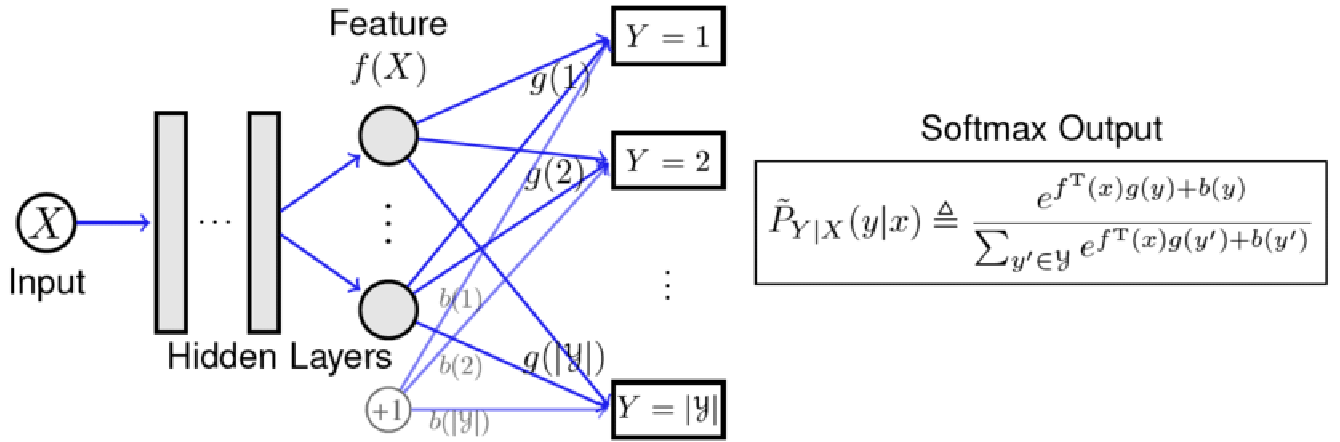

Figure 2.2: Classification example

Figure 2.2 shows another example of a classification network. This network has multiple hidden layers and an output layer with multiple classes from the set Y. To output the probabilities of an item being in each of the classes, the values from the output neurons are passed through the $softmax$ function.

Neurons also require an activation function to introduce non-linearity into the model. Without non-linearity, the multiplication of weights from each layer would result in a function that can be represented as a single linear transformation, defeating the purpose of having multiple layers in the architecture. A commonly used activation function is ReLU $max(0, x)$.

To teach the model, the weights of the neural network can be adjusted depending on the accuracy of the predictions by methods called back-propagation and gradient descend. Back-propagation calculates the impact of each weight on the total error using derivation. The error is calculated using a loss function such as negative log likelihood

$$\text{NLL} = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

, where N is the number of items, C is the number of classes, y_{ij} is 0 or 1 depending on whether class j is correct for item i and $\hat{y}_{ij}$ is the predicted class. Then, the gradient descend method is used to adjust the weights and potentialy lower the error in the next iteration. This process can be repeated multiple times, until the model can make predictions well enough. The process is fittingly called training. An important thing

to keep in mind is that a trained model needs to be tested on unseen data since it can memorize all the data it was trained on. Overfitting is a term describing a model that learns the patterns of the training dataset near perfectly without being able to keep the same performance when given unseen data. In machine learning, one is interested in generalization, the model needs to find patterns that are common for all the relevant data it can encounter.

One epoch is described as one iteration of training where the model has gone through all the training data once. The number of epochs or how long the model must be trained depends heavily on the parameters and how complex the patterns are. Training can be stopped after a set number of epochs, if the information loss is nearing zero or when the loss plateaued indicating that the model cannot improve.

2.2 Graphs

The following gives a short introduction to what a graph is and the graph terminology used in this work.

A graph G is a pair (V, E), where V is a set of vertices and E is a set of edges between the vertices $E \subseteq \{(v, u)|v, u \in V\}$ [17]. Two vertices with an edge between them are adjacent or connected. In an undirected graph an edge implies that the connection is bidirectional, meaning (v, u) and (u, v) are equivalent. Figure 2.3 demonstrates the difference between an undirected graph (right) and a directed graph (right).

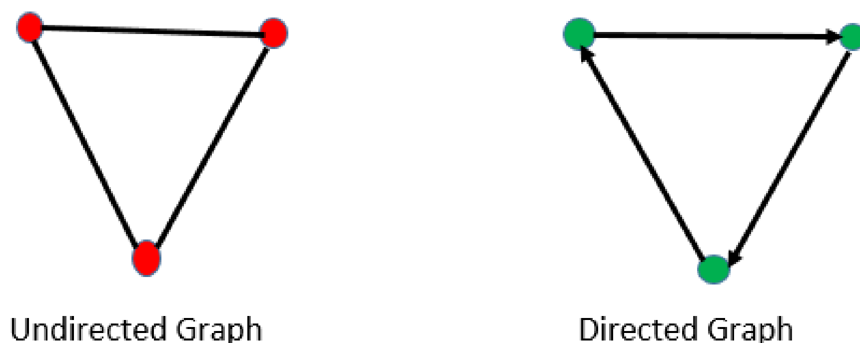

Figure 2.3: The difference between a directed and an undirected graph
Source: https://byjus.com/maths/graph-theory/

For graphs with weighted edges, let $W()$ be a weight function on the edges such that $W(e) \rightarrow \mathbb{Q}^+$

In this work we convert graphs to their line graph representation. The line graph is a complement of the original graph that turns each edge to a vertex and connects the vertices if they shared a vertex in the original graph. A simple example of a graph and its line graph can be seen in figure 2.4.

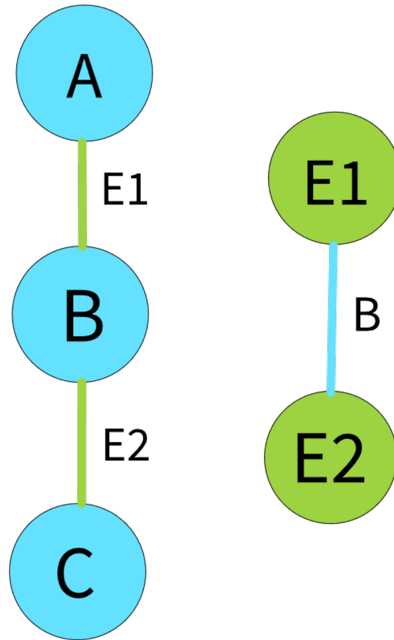

Figure 2.4: Original graph (left) and its line graph (right)

Note that the line graph willl always have the number of nodes equal to the number of edges in the original graph while, the number of edges in the line graph can vary and can be higher than the number of the original nodes.

In general, for graphs we use following notations:

$$nodes/vertices : v \in V,$$

$$edges : e_{v,w} \in E | v, w \in V$$

$$neighbourhoods : N(v) \subseteq \{w, (v, w) \in E\}.$$

2.3 Graph Neural Networks

Graph Neural Network (GNN) is a relatively young subclass of neural networks designed specifically for solving graph related problems, and has been a growing topic of research in the last couple of years. GNNs in general can be used for three purposes: graph prediction, node prediction and edge prediction. Graph prediction can be used to find graphs with desired properties, for example, to determine if a graph has cycles in it. Similarly, node and edge prediction is applicable to classification problems, such as identifying nodes and edges that are part of a cycle.

The application of graph prediction can be seen in bioinformatics. Ramachandran et al. have used graph convolutional network to predict how different proteins interact with each other [21]. Node-level prediction has found application in traffic forecasting [31]. Edge or link prediction models can be seen in the context of social networks with applications such as friends recommendation [1]. Another popular field for GNN research is Combinatorial Optimization (CO), which leads us to the well known problems such as the Maximum Weighted Matching (MWM) and Maximal Independent Set (MIS).

Brusca et al. [4] showed a self-training GNN for MIS.

Schuetz et al. made an unsupervised GNN [24] for solving MIS and Angelini and Ricci-Tersenghi [2] compared its performance to greedy algorithms and reported some problems with GNNs performance.

The reason MIS solving is mentioned is because the problem is to some degree related to MWM, as algorithmic problems often are. There are concrete examples in the Methodology and Data chapter. MIS is also more often a subject for research, while research on MWM is not that common. There are, however, some examples such as Wu and Li trying to solve MWM using deep reinforcement learning [30]. In this work they reported that their model outperforms state-of-the-art algorithms.

GNNs are based on the same concepts as simple neural networks, but with some modifications to better suit graph, node and edge recognition problems. A simple GNN can use a fully connected layer on each node and edge, as well as on the whole graph at once to generate a new numericatal representation for each element called embedding. In the following figure 2.5 one can see a simple overview of how the graph's information is passed from one layer to another, where V denotes nodes, E edges, U graph-level features and update function is a neural network layer.

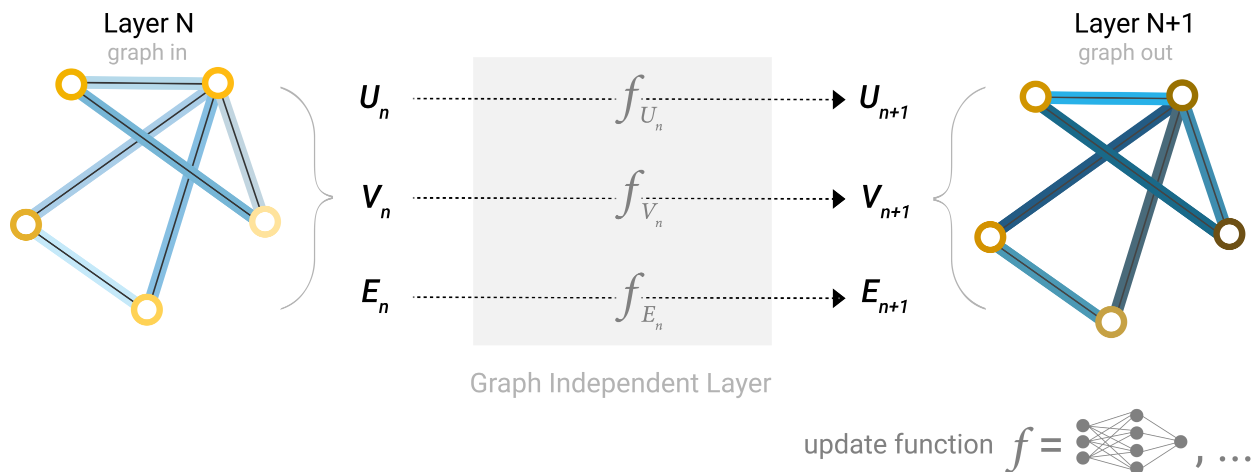

Figure 2.5: GNN example
Source: https://distill.pub/2021/gnn-intro/

For a simple node classification example, assume a graph with some nodes, where each node has two numerical features $f1$ and $f2$. Assume a GNN that uses a simple fully connected network, where the input layer has two neurons, the middle layer has five and the output layer has two. For each node, the input layer reads the two features. Values of these features are then multiplied by the weights between each input neuron and each neuron in the next layer. One can express it as a matrix multiplication:

$$\begin{pmatrix} f1 & f2 \end{pmatrix} \times \begin{pmatrix} w11 & w12 & w13 & w14 & w15 \\ w21 & w22 & w23 & w24 & w25 \end{pmatrix} = \begin{pmatrix} n1 & n2 & n3 & n4 & n5 \end{pmatrix}$$

Where the $f1, f2$ are node features and $w11$ to $w15$ are the five weights between the $f1$ input neuron and all the neurons in the next layers.

The information is then similarly passed from the middle layer to the output layer. The initial values of the node features have now been changed by the weights between the neurons, resulting in two output features. The GNN has now produced an embedding for each node. Yet, the embeddings by themselves are not enough. In this example, the GNN does not utilize the structure of the graph and the embedding of each node is not affected by its neighbours.

If one wants to classify the nodes based on the node features only, then it can be done by using a standard neural network. However, for graphs, one is interested not only in the isolated node features, but in the nodes' relation to their neighbours. Additionally, graphs are not guaranteed to have any features for nodes and edges.

What if one wants to use edges for classification of the nodes? In that case GNN can use a pooling method. Pooling uses an aggregation function such as summation to sum together all the edge embeddings that a node has, and produce a new embedding. Alternatively, one can use neighbouring nodes for the same purpose, which leads to the specific type of GNNs.

Graph Convolution Network (GCN) is one of the subclasses of GNNs with some resemblance to the Convolutional Neural Network (CNN)s used in image recognition. GCN uses message passing to make use of the graphs' structure. For nodes, message passing gathers neighbouring node embeddings, aggregates the embeddings and then passes them through the network. The following figure 2.6 shows how features from the neighbouring nodes are aggregated and passed to the next layer.

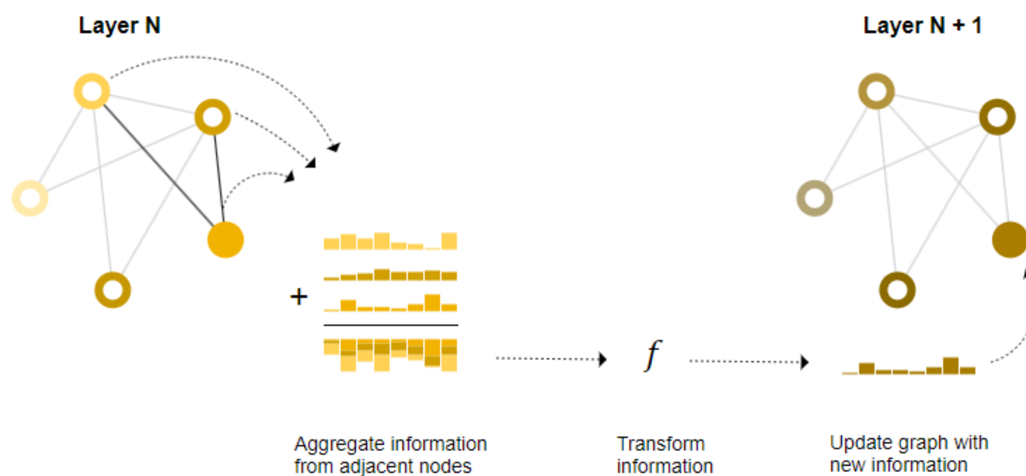

Figure 2.6: Message passing example
Source: https://distill.pub/2021/gnn-intro/

Adding more layers increases the depth of the neighbourhood used. Meaning if a model has two layers, when passing through the second layer the embeddings of the neighbours are already an aggregation of its neighbours. This allows the GNNs to utilize graph structure in a way that other neural networks cannot.

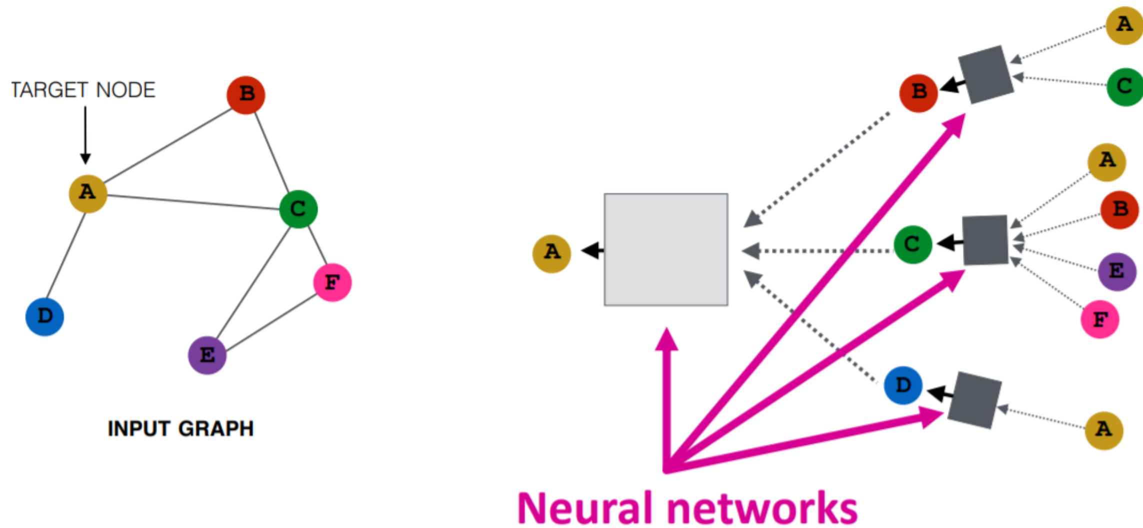

Figure 2.7: GCN example
Source: https://www.datacamp.com/tutorial/comprehensive-introduction-graph-neural-networks-gnns-tutorial

Figure 2.7 shows how two graph convolution layers affect a single node A.

2.4 Implementation, training and hyperparameters

We have now gone through how GNNs work from a high level view. However, it remains to create a GNN as well as to train it. In this work we use Pytorch Geometric (PyG) library [19] that provides both the impementation for the various GNN architectures and the tools to train the models.

2.4.1 Implementation

The code snippet below 2.1 demonstrates how to create a simple GCN by inheriting the base class *torch.nn.Module*.

Listing 2.1: Creating GCN with Pytorch

```
1 import torch
2 import torch.nn.functional as F
3 from torch_geometric.nn import GCNConv, Linear
4
5 class MyGCN(torch.nn.Module):
6     def __init__(self):
```

```
        super().__init__()
        self.conv1 = GCNConv(1, 64) # 1 feature per each node
        self.conv2 = GCNConv(64, 64)
        self.lin = Linear(64, 2) # 2 classes

    def forward(self, graph):
        x, edge_index, edge_weight = graph.x, graph.edge_index,
           ↪ graph.edge_weight

        x = self.conv1(x, edge_index, edge_weight)
        x = F.relu(x)

        x = self.conv2(x, edge_index, edge_weight)
        x = F.relu(x)

        x = self.lin(x)
        return F.log_softmax(x, dim=1)
```

A module in Pytorch can be a part of a larger network or a whole network in this case. One can add layers to the network by assigning other modules within the initializer function. Here, we add three layers. The first layer, $self.conv1 = GCNConv(1, 64)$ is the input layer with one input neuron and 64 output neurons. This layer will require all nodes to have exactly one feature. To connect the $conv1$ layer to the second one, $conv2$, they must have a matching number of output and input neurons (64 in this example) respectively. The output module $self.lin = Linear(64, 2)$, is a simple fully connected layer that maps output neurons from the $conv2$ to the two neurons representing the two possible predictions. For example, whether a node should or should not be in the solution. In the $forward$ function, one has to define the order in which the information will flow through the network. First, one has to read the input from the graph. The x represents the array of node features. In this case, the array is of size one. The $edge_index$ and $edge_weight$ represent the edges and their weights respectively. The $torch.nn.functional$ library provides the tool for mathematical operations and is used to apply the $F.relu$ activation function to the neurons as well as $F.log_softmax$ to the output. The $relu = max(0, x)$ activation function serves the purpose of adding non-linearity to the network, enabling it to form complex patterns. The $F.log_softmax$ function maps the output values into the desired range for measuring the error. Which function is used depends on how the error is calculated.

2.4.2 Training

To train the model described above, the following procedure can be done, demonstrated in the next listing 2.2.

Listing 2.2: Training GCN with Pytorch

```
import torch
import torch.nn.functional as F
from torch_geometric.loader import DataLoader

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
loader = DataLoader(dataset, shuffle=True)
optimizer = torch.optim.Adam(list(model.parameters()), lr=0.0005)

for epoch in range(100):
    for graph in loader:
        optimizer.zero_grad()

        graph = graph.to(device)
        out = model(graph).to(device)

        loss = F.nll_loss(out, graph.y)
        loss.backward()

        optimizer.step()
```

First, the *DataLoader* and optimizer are initialized. The *DataLoader* is responsible for managing and preproccessing the data. In this case, it only shuffles the dataset items randomly. The optimizer is used to preform the gradient descent on the models weights. Adam [1] is a commonly used optimizer that has several parameters such as learning rate (lr) that affect the training process and how the weights are updated. These parameters in the context of the model are called the model's hyperparameters. With dataloader and optimizer ready, the main training loop starts. For each epoch or iteration, we retrieve the next batch of data from the *dataloader*. Before updating weights, *optimizer.zero_grad*() is called to make sure that the optimizer's gradients are reset. Then we pass the graph to the GPU device if possible to speed up the calculations. The graph is then passed through the model with the output mapped using *log_softmax* within the model's forward function. The *log_softmax* mapping produces the logarithmic probabilities required by the *nll_loss* function to compute the error or information loss made from the model's output in regard to the correct answers contained in *graph.y*. The error is then propagated backward by calling *loss.backward*(), determining the impact each weight made and then *optimizer.step*() updates the weights based on the current hyperparameters.

2.4.3 Hyperparameters

The term "hyperparameter" refers to the parameters used to adjust the training process and the architecture of the network, while "parameters" refer to the weights between neurons. In this work we test several hyperparameters that can be further split into categories.

The training and Adam optimizer related hyperparameters include:

- Epochs - the number of training iterations. The longer a model is trained the less errors it should make, given the sufficient complexity of the model.
- Learning rate - by how much the weights of the model should be adjusted after each iteration. Small steps have a risk of a model getting stuck in a place where the changes do not improve the results, while a significantly different configuration of weights that gives better results might exist. Small steps also might require to train the model for longer to reach the desired level of performance. Too large learning rates have a risk of overstepping the better weight configurations.
- Mini-Batching size - how many graphs are passed through the network before next weight adjustment. Passing data in batches can help to smooth the learning curve and reduce the risk of overfitting.
- Class weights - how much should the model focus on a given class. Class weights can help with data imbalance where one class occurs more often than others. For example if 99% of objects belong to one class, but the model needs to focus on finding the other 1%. In our case the classes for the edges are:
  - 0 = is NOT in the solution.
  - 1 = is in the solution.
- Weight decay - used to keep GNN weight values relatively small and decrease chances of the overfitting.

The architecture related hyperparameters affect the capability of the network to learn complex patterns.

- Network width - neurons in the layers.
- Network depth - amount of layers. In case of GCNs, the depth of the network also defines the radius of the aggregated node neighbourhood.

The aggregation of neighbours is the key feature of the GNNs and there are multiple functions that can be used. GNN specific hyperparameters:

- Aggregation function - the features of the neighbouring nodes can be aggregated by using different methods, including summation, maximum, minimum, multiplication and division.

Additionally one can experiment with adding extra features to the nodes during pre-processing or augmenting data. Examples of such features for the weighted graph are:

- Degree - how many neighbours a node has
- Sum of the weights.

Such features can help the model to find new patterns.

The hyperparameter search helps to fit the model specifically for the desired task. In our case the model needs to learn how to solve Maximum Weighted Matching (MWM) on graphs.

2.5 Maximum Weighted Matching and Combinatorial Optimization

Combinatorial Optimization (CO) is a field of study that covers problems that require finding a subset or a combination of some set of elements that fulfills certain requirements. An example of such a problem would be Maximum Weighted Matching MWM and Maximum Independent Set MIS.

The MWM problem asks to find a subset of edges in a weighted graph, such that the sum of weights is maximized, and each vertex appears only in one edge of that subset. More formally, given an undirected graph G with a set of vertices V and a set of weighted edges E, maximize total weight for set $S \subseteq E$, such that none of the edges in S share vertices. In this work, the edges that are selected to be in the solution set will sometimes be referred to as "picked" and the ones that are not in the set "dropped".

The MWM problem is relatively simple compared to many others. It belongs to the field of CO and there are optimal algorithms for it that run in $O(|V|^2|E|)$ time, such as the blossom algorithm [6]. Blossom algorithm is based on the blossom method for finding augmenting paths and the primal-dual method for finding a matching of maximum weight, both due to Jack Edmonds [6].

Optimal algorithms, however, can still be rather slow when working with large datasets, and in some cases an approximation algorithm may be more suitable as a solution if the exact answer is not critically important for the MWM problem. The most simple example is a greedy algorithm that sorts all the edges by their weight and selects them in descending order. This is neither the only nor the best approximation algorithm, but rather a fast and simple one.

Algorithm 1 Greedy algorithm for Maximum Weighted Matching

```
G(V,E)
S = ∅
sortedE = descendingSort(E)
while sortedE ≠ ∅ do
    e = first element in sortedE
    if e not adjacent to any edge in S then
        add e to S
        remove e from sortedE
    end if
end while
```

The running time of the Greedy algorithm is bound by the sorting, since the While loop iterates through the edges once and runs in $O(|E|)$. The fastest sorting runs in $O(|E|log(|E|))$.

To put MWM in context, real world applications of matching algorithms can be found in scheduling problems such as transportation cost minimization [10], but are also often used as a part of larger algorithms.

Another problem that will be slightly touched is the Maximal Independent Set (MIS). MIS asks to find a subset of nodes such that no two nodes have an edge between them, and the total number of nodes is as large as possible. There is a similar problem Maximum Weighted Independent Set (MWIS) which asks us to find an independent set in graph with weighted nodes that gives the largest total weight sum. MWIS is partially relevant as we use line graph transformation that essentially turns the original MWM problem into MWIS.

Chapter 3

Research Methodology and Data

This chapter focuses on demonstrating and explaining how the model was designed and trained. An overview of the design and architecture is given. Relevant training hyperparameters are discussed. Datasets are presented and analysed. Challenges are explained together with changes and adjustments that were made during the process. Finally, goals and criteria for the expected results are defined.

To reiterate, the current problem is to find out if a Graph Neural Network (GNN) based method can compete at solving the Maximum Weighted Matching (MWM) problem compared to other approximation methods such as the greedy algorithm.

3.1 Challenges

It is common to think of a neural network as a black box that takes in data as input and outputs some predictions. In our case, the data is a graph consisting of vertices connected by weighted edges and the expected answer should be pairs of adjacent vertices that are matched together or, equivalently, a list of the corresponding edges.

As mentioned, the idea behind supervised learning is to give the correct answers to a model, so it can, by trial and error, learn from it. These answers are not included in the initial datasets, so the optimal solutions need to be calculated. To find the optimal solution, the Blossom algorithm implemented by Rantwijk in Python was used [27]. This does, however, point out an important weakness of the supervised approach. Obviously, a model needs to be able to handle graphs of different sizes, and it is the large ones

that are most interesting. Finding an optimal solution for algorithmic problems for the large graphs can be time-consuming. At the same time, a model needs as much data as possible to learn, leading to multiple large graphs consuming too much time during training. This poses a question of whether it is possible for the model to learn from small and medium-sized graphs that are not as time-consuming and apply the learned patterns to solve larger graphs.

Another important aspect that must be taken into consideration is that it is unlikely for the model to fully follow the restrictions of the problem. As an example, the model might decide to match the same node to two neighbors at the same time, which should not be allowed.

3.2 Main Idea

To summarize, given a weighted undirected graph, the GNN must predict a valid subset of non-adjacent edges that maximize the total weight. The GNN must teke a graph as an input. The size of the input layer will be equal to the amount of features nodes have. Initially, nodes do not have any features and, in that case, all nodes can be assigned a single feature with $value = 1$. Later, however, we experiment with adding additional precomputed features. The output layer will be of size two, representing the two possible outcomes for whether each edge should be in the matching or not.

To check that a solution is valid, instead of picking all the edges with a probability higher than 50%, the edges can be sorted by their probabilities and greedily picked in that order as long as they don't break the validity of the solution. It might sound illogical to use a greedy algorithm on the models' output to beat the normal greedy algorithm with weights, but the fact that edges are now sorted not by their weight, but a score decided by the GNN does make a difference, since it can take multiple features in consideration when assigning probabilities. Some of these features come naturally from the structure of the graph, and some we can manually add during preprocessing. This approach does imply that the expected running time on the GNN will be higher than that of a greedy algorithm due to the fact that the sorting of edges still remains, but on top of that, the graph must be processed by the model.

Neural Networks have several hyperparameters and methods that can be tuned to make a model better suited for the task. With unlimited resources and time, one could

search for the best combination of the hyperparameters by trying as many combinations as possible and training a separate model for each combination. Then choose the one with the best performance. However, this is a time-consuming procedure, therefore, in the current work, the exploration of hyperparameters has been narrowed down to the ones that seemed most important and indicated a positive impact during early experiments. For training, Adam optimizer was used [13]. The optimizer is used to adjust the weights of the network during training. Adam optimizer has several hyperparameters that can be useful to achieve better results and is one of the commonly used optimizers.

During the experiments the following hyperparameters were tested:

- Epochs - the number of training iterations
- Learning rate - by how much the weights of the model should be adjusted after each iteration
- Network depth - amount of layers
- Network width - neurons in the layers
- Class weights - how much should the model focus on a given class. Classes being:
  - 0 = is NOT in the solution.
  - 1 = is in the solution.
- Weight decay - used to keep GNN weight values relatively small and decrease chances of the overfitting.
- Additionally other methods were tried such as augmenting data (adding extra features to the nodes during preproccessing):
  - Degree - how many neighbours a node has
  - Weight relative to the sum of neighbours.
  - Weight difference from the sum of the neighbours.
  - Sum of the weights.
  - 1st largest weight, 2nd largest weight.
- Aggregation function - for the features of the neighbouring nodes

Experimenting with the hyperparameters allows one to find the configuration for the model that balances it between overgeneralizing its prediction and memorizing the training dataset without being able to make accurate predictions based on the unseen data. Low learning rates make the training process more time-consuming, while large learning rates make weight adjustments that can overstep weight values that potentially could

have improved performance. The depth and the width of the network are responsible for the level of complexity the model can learn. However, one does not need an overly complex model, since it both slows the model down and makes it susceptible to memorizing the dataset. Class weights can help with the distribution of data. If nodes in a graph have multiple neighbours, a good portion of the edges will not be in the solution. The model might be affected by the dominant number of edges that are not part of the solution to the point where it is not focusing enough on finding the ones that are in the solution. Adding extra features might help the model to find new patterns. The aggregation function is the only parameter specific to the GNN and is responsible for the gathering of information about the neighboring nodes.

3.3 Data

The model should be capable of solving any graph relevant to the MWM problem. For a graph to be relevant for the task, it is only required to be undirected and have weighted edges. Although one can also use unweighted graphs with all edge weights set to one.

For training and experimenting with the GNN, the MNIST dataset was used [5]. The dataset consists of 70000 relatively small graphs with 70 nodes and 564 edges on average. Graphs also include features for the edges representing distances between nodes. These features are used as the weights for the problem. Although the graphs in this dataset are relatively small, they have some fitting qualities, such as nodes having multiple neighbors, creating more possible ways to match the nodes. The small size also makes it less time-consuming to train the models and try different approaches, as well as to show whether a GNN trained on smaller graphs can transfer its knowledge to larger graphs.

Training and testing the model only on MNIST graphs does not cover all the various graphs that can occur. Therefore, the model had to be tested on other graphs that have different structures and weight distributions. Models trained exclusively on the MNIST dataset did not perform as well on other graphs as one might have expected. Therefore, another dataset was made to cover a larger variety of graphs. The dataset was a collection of random graphs from SuiteSparse database that ranged between 100 and 10000 nodes. As described in more detail in the 3.4.4 Data preprocessing section, some of the graphs did not have edge weights. In these cases, random weights were generated. The graphs were collected by using the following filter in Python that downloads a list of the graphs that fit the set criteria:

,

```python
import ssgetpy as ss
dataset = ss.search(
        limit = 10000,
        kind='Weighted',
        rowbounds = (50,10000),
        colbounds = (50,10000),
        nzbounds = (10,1000000))
```

Including truly large graphs in training caused problems with insufficient memory and too long training time, but the final model was still tested on the larger graphs than ones used in training. The graphs were taken from different datasets described in table 3.1.

Table 3.1: Names and sizes of graphs used for testing

Graph	Nodes	Edges
G22	2 000	40 000
Cage9	3 500	38 000
California	5 000	20 000
G55	5 000	25 000
Cage10	11 000	140 000
as-22july06	22 000	96 000
dictionary28	39 000	178 000

Small handcrafted graphs as well as modified versions of the existing graphs, demonstrated in the 4.3.1 Weakness of the greedy algorithm, were used to test if GNN can at least beat the cases specifically made to put greedy approach in a disadvantage.

3.4 Architecture

During the experiments, the architecture of the GNN model itself had mostly minor changes, but two rather different graph preprocessing steps were used. In both cases we use GCN layers (GCNConv) [14]. The GCNConv layer fits well for the purpose of this task due to it's message passing utilizing the edge weights that are the crucial part for solving MWM. The input layer has the same size as the number of node features. Then the graph convolutional layers are added, the depth and width of the middle layers vary depending on the temporary training results. For the output layer, a logarithmic softmax function is applied to obtain the log-probabilities required to calculate the negative log likelihood loss. These probabilities can then be translated into normal probabilities.

The next code snippet 3.1 shows an example of the Python implementation for the line graph GCN. GCNConv implements a single layer of the network. The input layer is the

size of the number of vertex features (5 in this example). The input layer connects to 640 neuron wide GCNConv layers. The last layer outputs predictions for each vertex through Linear layer with one neuron for each possible outcome. The first neuron represents the probability of the edge NOT being in the matching and the second represents the opposite. The "forward" method is used to define how the data flows from one layer to another. The "relu" function $f(x) = max(0, x)$ is a commonly used activation function for neural networks. The purpose of the activation function is to add non-linearity to the mode.

Listing 3.1: Pytorch GCN for the line graph

```
1  class MyGCN(torch.nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.conv1 = GCNConv(5, 640) # 5 features per each node,
               ↪ initialy 1
5          self.conv2 = GCNConv(640, 640)
6          self.lin = Linear(640, 2) # 2 classes
7
8      def forward(self, graph):
9          x, edge_index, edge_weight = graph.x, graph.edge_index,
               ↪ graph.edge_weight
10
11         x = self.conv1(x, edge_index, edge_weight)
12         x = F.relu(x)
13
14         x = self.conv2(x, edge_index, edge_weight)
15         x = F.relu(x)
16
17         x = self.lin(x)
18         return F.log_softmax(x, dim=1)
```

We have also experimented with adding skip connections to the architecture. Skip connections are used to let information from one layer jump over some layers to reach the deeper ones. This method helps to emphasize the importance of earlier layers as their impact tends to vanish in deeper networks. Additionally, skip connections help the network to learn the identity function $f(x) = x$, which can be helpful in cases where complex patterns are not needed. The listing 3.2 shows how one can implement skip connection between first layer $self.conv1$ and the output layer $self.lin$

Listing 3.2: Skip connection example

```
1  def forward(self, data):
2          x, edge_index = data.x, data.edge_index
3
4          x = self.conv1(x, edge_index)
5          identity = F.relu(x)
6
7          x = self.conv2(identity, edge_index)
8          x = F.relu(x)
9
10         x = self.conv3(x, edge_index)
11         x = F.relu(x)
12
13         x = torch.cat((x,identity),1)
```

```
14        x = self.lin(x)
15
16        return F.log_softmax(x, dim=1)
```

Now, let us go through the process of a graph being matched using the GNN.

3.4.1 Model pipeline

As mentioned earlier, one cannot fully trust the model to satisfy the restrictions of the problem. What is meant by the restrictions of the problem is the fact that every node can only be matched once. This is controlled by sorting the probabilities of the model's outputs in descending order and adding matches to the final solution one by one, skipping matches that already have matched nodes. Another potential problem can arise at the end when some of the nodes that can be matched were not matched at all by the model. There is a chance of that happening, since only matches with a high enough probability are used. As a starting point, the probability threshold is 50% and above. Two precautions are taken to ensure nothing is left to waste. First, after the model is done, the potential remainder of the graph is fed to the model again. For a model, this would essentially be a new graph, and it may happen that the model manages to match the remainder given the new context. This process can be repeated several times and the break condition is if 0 matches were made in the last iteration. At that point, if anything is left, which hopefully is a small fraction of the initial graph, it can be solved using standard greedy or any other approximation algorithm, for that matter.

Passing the graph, and its consecutive remainders multiple times through the model can also be helpful to deal with the cases where one of the matches that was chosen may strongly affect how the rest of the solution will look like. This leads to the main weakness of the supervised approach. The optimal solution can be completely different if even one edge is picked or not picked by a mistake. This problem can be mitigated by picking one match at a time, updating the features and passing the graph through the model again, but it would be too time-consuming.

3.4.2 Line Graph Approach

The line graph approach was the first attempt at using a simple GNN, which later turned out to be overly time-consuming for larger graphs to be worth further experiments. It

Algorithm 2 Model Pipeline

```
inputGraph
S = ∅
g = preproccess(inputGraph)
while g.hasUnmatchNodes do
    edgeProbabilities = model.makePrediction(g)
    matches = greedyPickByProb(edgeProbabilities)
    g = g.remove(matches)
    S.add(matches)
    if matches == ∅ then
        matches = greedyPickByWeight(g.edgeWeights)
        g = g.remove(matches)
        S.add(matches)
    end if
end while
return S
```

did, however, give some useful insight as well as shown to be a proof of concept. The 2.2 Graphs section gives the description and an example of a line graph.

Convertion to a line graph makes the architecture of the GNN model simpler. Instead of needing to ask a model to make predictions about whether each connected pair of nodes should be matched together, the line graph model can now output the probability of a node being in the matching directly, since the node now represents two connected nodes from the original graph. This also transforms the problem itself. From the perspective of the model, it is now trying to solve MWIS. GNNs for MIS have been studied before. Nouranizadeh et. al. demonstrated a pooling method for solving MWIS [18] so it would be reasonable to expect the model to be able to perform relatively well.

Unfortunately, large graphs can explode in size when converted to line graphs and are too time-consuming in such cases. Consider a graph from figure 3.3 with a node 0 in the center connected to three other nodes. The size of such a graph would be four nodes and three edges. In a line graph, that would result in a clique of three nodes with every node being connected all the other nodes. But if the node 0 had 10 neighbors instead, the line graph will result in a clique of 10 nodes and $1/2 * 10 * (10 - 1) = 45$ edges. While the number of nodes in a line graph will always be equal to the number of edges in the original graph, the number of edges can be significantly larger. Since the running time of solving MWM depends on the number of edges, there is a reason to try other approaches.

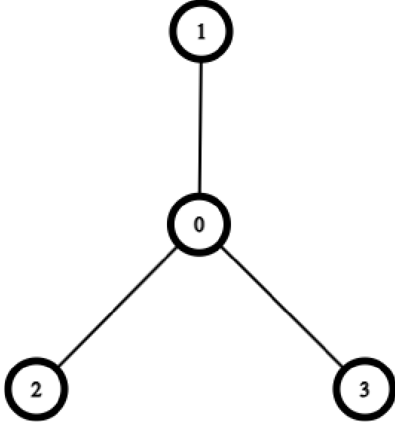

Figure 3.1: Original graph

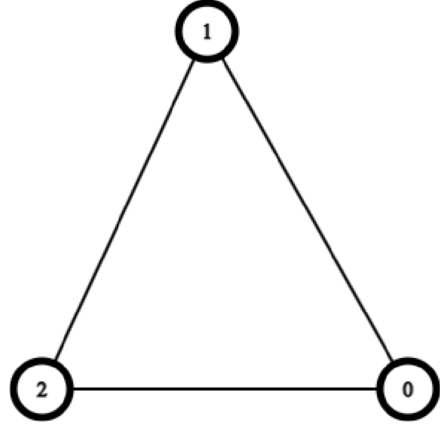

Figure 3.2: Line graph

Figure 3.3: Line graph convertion example

3.4.3 Edge Classification Approach

Another way of approaching this problem is to work directly on the original graphs instead of transforming them into line graphs. However, the model itself only returns some numerical representation of the nodes in the graph. To get the predictions for the edges one can add a classifier module to the existing GNN. The purpose of this module is to predict if a pair of two nodes will be in the matching based on their embedding. The first part of the model produces embedding for each node and then, for each edge, embeddings of both nodes are put together and passed to the classifier. Finally, the model outputs probabilities of all the edges being in the matching. This approach eliminates the need for the line graph conversion which often leads to larger graphs and slower running time.

The next code snippet 3.3 shows an example of the Python implementation for the edge classifier network using Pytorch libraries. The MyGCNEdge class is responsible for embedding the vertices of the graph. The input layer is the size of the number of the vertex features (5 in this example). The input layer connects to 640 neuron wide GCNConv layers and outputs feature array of size 80 for each vertex through Linear layer. Then EdgeClassifier class can be used to create the edge embeddings using embedEdges method that concatenates 80 features from two endpoints of an edge together using *torch.cat*(). Each edge is then passed through an input layer with $80 + 80$ total features and further through fully connected Linear layers. The output layer is of size two with

the logsoftmax function at the end to get the values required for the loss calculation, same as it is in the line graph model.

Listing 3.3: Pytorch GCN for the edge classification

```

class MyGCNEdge(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = GCNConv(5, 640)
        self.conv2 = GCNConv(640, 640)
        self.embed = Linear(640, 80) # 80 output features per node

    def forward(self, data):
        x, edge_index, edge_weight = data.x, data.edge_index,
            ↪ data.edge_weight

        x = self.conv1(x, edge_index, edge_weight)
        identity = F.relu(x)

        x = self.conv2(x, edge_index, edge_weight)
        x = F.relu(x)

        x = self.embed(x)
        return x

class EdgeClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.lin1 = Linear(160, 320) # 80 * 2 features since input is
            ↪ 2 nodes.
        self.lin2 = Linear(320, 2)

    def embedEdges(self, nodeEmbed, graph):
        x_src, x_dst = nodeEmbed[graph.edge_index[0]],
            ↪ nodeEmbed[graph.edge_index[1]]
        edgeEmbed = torch.cat([x_src, x_dst], dim=-1)
        return edgeEmbed

    def forward(self, edgeEmbed):
        x = self.lin1(edgeEmbed)
        x = F.relu(x)

        x = self.lin2(x)
        return F.log_softmax(x, dim=1)
```

3.4.4 Data preprocessing

Not all graphs found in the database are fit for the training as is, and the ones that fit perfectly are few. Machine learning models often require some preprocessing of data before they can be used. Conversion to the line graph is one example of such a preprocessing. One might also need to augment the initial data to improve model performance. Finally, machine learning models can be vulnerable to large numerical variations in the data, such as variation of the edge weights.

For MWM, the edge weight is the key feature that is required for the graphs to be viable. To ensure we have enough graphs to train on, and the graphs cover a variety

of structures, different data source might be used. Some of the encountered graphs might lack weights and will have to receive randomly generated weights. We assign the random weights using even distribution in the range between 0 and 1. All the graphs with preexisting weights are also adjusted to be in the same range by shifting the weights to a positive range if any negative weights were detected and then dividing by the largest weight in the dataset. The weights could have been in any range, but the important thing is to have consistent weights for all the graphs.

Another viable preprocessing step is node feature augmentation. One can use the edge weight to precompute potentialy helpfull information such as weight sum of all the edges of each node. This requires different approaches for line graph and edge prediction approaches, since in a line graph, the vertex represents an edge from the original graph.

One node feature does remain in common: the degree of the nodes or how many neighbours each node has. Although the degrees have to be calculated separately for the original and the line graph, since degree of the line graph nodes represents the number of adjacent edges of an edge in the original graph.

Line graph's node features:

1. Weight relative to the sum of neighbours. For each vertex v:

$$Wrel = v.weight \div (\sum_{n \in N(v)} n.weight)$$

2. Weight difference from the sum of the neighbours. For each vertex v:

$$Wdiff = v.weight - (\sum_{n \in N(v)} n.weight)$$

3. Sum of the neighbouring node weights For each vertex v:

$$Wsum = (\sum_{n \in N(v)} n.weight)$$

The motivation for adding these features is based on the idea that precalculating the relations between the edges and the neighbouring edges might allow the GCN to find patterns easier. For example, in the line graph, if a node n has weight $W(n) = 10$ and the sum of neighbouring nodes is $Wsum < 10$, then $Wrel > 1$ and any combination of two

neighbouring nodes has total weight < 10 and therefore node n, that represents the edge in the original graph, should be in the solution. This is a simple case that the GNN might find without additional features, but it serves as an example. It might be unintuitive how these features can be helpful, but one of the GNN's functions is to find difficult patterns, and it is worth testing such features even if their effect is not apparent. The weight of the edges is also the only input feature available in this task, so the additional features are dependent on it.

For the original graph's node features, some adjustments are made since nodes do not have any weights assigned to them, unlike the line graph:

1. Sum of the weights of all the outgoing edges. For each vertex v:

$$Wsum = (\sum_{j \in N(v)} e_{v,j}.weight)$$

2. Wsum relative to the Wsum of neighbours. For each vertex v:

$$Wrel = Wsum \div (\sum_{n \in N(v)} n.Wsum)$$

3. Wsum difference from the Wsum of the neighbours. For each vertex v:

$$Wdiff = Wsum - (\sum_{n \in N(v)} n.Wsum)$$

3.5 Result Validation

Before looking at the results, it is important to decide how to evaluate the results properly and what is important for the problem at hand:

1. Time - how long an algorithm took to produce an answer. For the GNN model, this includes model specific preprocessing such as adding additional features as well as finishing the potential remainder of the graph with a greedy algorithm.

2. Correctness - is the answer correct? In the case of MWM the total weight acquired would be the measurement of the quality of the solution. It is unlikely that the GNN model can find an optimal solution for more complex problems, so it is reasonable to look at how close the GNN comes to the optimal solution.

3. Remainder - a portion of the final solutions' weight that came directly from the models' predictions. Due to the way the program is set up, if the model does not match the whole graph, the rest of the graph will need to be finished somehow. This is done by running a normal greedy algorithm at the end if anything is left. Because of that, there can be cases where the model does nothing and by default achieves 100% result compared to what a normal greedy algorithm would. Therefore, the portion of the weights obtained directly by the model needs to be evaluated as well.

3.5.1 Sanity checks

It might happen that during the project something goes wrong with the mode. Therefore, it is common to perform sanity checks to see if a model's inputs and outputs still make sense.

The following "sanity checks" were done:

1. To ensure the model and data are functioning correctly, we trained a model on a small dataset for a long time and tested it on the same data. The model should be able to memorize the provided graphs and solve the problem near perfect then.
2. Comparisons to random matchings were done. Before the training, the model has random weights and essentially will behave as if it is picking edges at random, but after some training the model is expected to start improving and perform better than a random matching would.
3. The model was tested on small handcrafted graphs that were made specifically to put the greedy algorithm at a disadvantage. If the GNN model manages to solve such cases that indicates that the model is learning some of the patterns.

3.6 Expected Results

3.6.1 Other observations

One important observation was made during the analysis of the data that is worth mentioning. The difference between the optimal solution and the greedy is, in most cases,

rather small. For the MNIST dataset and the majority of the graphs used from the Suite Sparse Matrix Collection, the greedy algorithm manages to reach 90+% of the optimal weight. Assigning random evenly distributed values to the edge weights also gives the same results. For the GNN this means that it will be rather difficult to beat the greedy algorithm since it gets fairly close to the optimal.

3.6.2 Accuracy and total weight

There exists limited amount of research specifically for for solving MWM using GNN. However, problems like MIS that are relatively similar to the MWM on the line graph can indicate that one could obtain similar results for this case as well. As discussed in the 2.3 Graph Neural Networks section, there are studies that show that GNNs are capable of solving CO problems and beating greedy algorithms, while others indicate that improvements are still needed for it to be worth using. Therefore, it is hard to forecast any results based on previous work. It is worth noting that the greedy algorithm is relatively simple. It is conceivable that a GNN model should be able to recognize more complex patterns that can be used to achieve better results. It is not expected for the model to be able to find an optimal solution since any Neural Network is a heuristic. The margin by which GNN can surpass the greedy solution is expected to be rather small, since from the data analysis it was observed that for the majority of graphs, the greedy algorithm performs rather well with above 90% of the optimal possible weight.

3.6.3 Time

The GNN model is a heuristic solver and gives an approximate answer. Therefore, a model should be noticeably faster than the exact algorithm, otherwise it would not be worth it. The time a model takes to solve one instance of a problem should be closer to that of a greedy algorithm and probably slightly longer due to preprocessing required, such as augmenting data with additional features. Naturally, time will also depend on the depth and the width of the network.

3.6.4 Remainder or model portion

Ideally, the model should be able to handle the full graph on its own, but as long as the model manages to improve the final result, it would still be a useful tool.

Chapter 4

Results

This chapter goes through the full process of making, training and testing the GNN step by step, from first iterations of the simple model to more fine tuned models and other methods that were tested. Reasoning and explanations for the choices and changes made during the process are also given along the way. Finally, best results that were achieved are presented.

4.1 Progress

4.1.1 Experimenting environment

The experiments have been done on a computer equiped with 11th Gen Intel(R) Core(TM) i7-11700K 3.60GHz 8-core CPU, 16 GB RAM and NVIDIA RTX 3080Ti graphics card. All the code for the GNN model was written in Python using Pytorch Geometric (PyG) library [19].

4.1.2 Simple line graph model

The first step was to try the simplest line graph model with most of the hyperparameters set to default and train it on a smaller scale using a small subset of 1000 graphs from the MNIST dataset, training it for 100 iterations and then evaluating its performance on 100 graphs from the evaluation dataset. The performance is evaluated based on the

total weight of the edges in the solution set compared to the greedy algorithm and the optimal if the results are close enough. Before using a large dataset that may take some time to process, it is worth confirming that the model works as intended. One also does not want to add complexity to the model without any reason. An unnecessary large and complex model affects the running time and has a higher chance of overfitting.

The first model had two layers with 64 neurons each. Remember that in a line graph every node represents an edge in the original graph. The first problem encountered was due to the majority of line graph nodes not being included in the matching. The model learned to set all nodes to be dropped and still get a rather high accuracy score, since the accuracy is measured by correctly classifying the nodes and not the total weight the classification produces. This was resolved by adding class weights as a parameter for the Adam optimizer during training. Class weights tell the model how important each class is, where the two possible classes stand for belonging to the solution and not. After testing the class weight distribution by increments of 0.1, the most effective result was achieved by assigning the nodes that are in the matching a value of 0.9 and the ones that are not 0.1. The problem seemed to be resolved, and the model's output was no longer one-sided. The model resulted on average with 55% of optimal weight possible when the tested on 100 graphs from the validation dateset. This was an expected result considering the small size of the current training dataset, but it at least showed that the model is gaining some knowledge compared to an untrained model, as well as a solution that randomly picks edges, where both resulted in 50-51% of the optimal solution.

The next step was to experiment with the learning rate parameter for the optimizer. The learning rate is used to decide by how much the models' weights are adjusted after each iteration. The high learning rates resulted in an unstable training with information loss jumping too much, which indicated that the model made overly large adjustments after each training iteration. The low learning rates did not give enough progress and made the training process too slow. The default value of 0.001 worked well as a middle ground, leaving the best result so far unchanged at 55%.

A model that barely outperforms a random solution is not very useful. Poor performance at this point was expected since the goal was to test if the model worked as expected and given a very limited training dataset. However, before adding more data and prolonging training, we can experiment with some of the parameters of the network to see if they make any impact.

4.1.3 Model improvements and data augmentation

At this stage, the model seemed to slowly improve and learn. Even with the current limit on training data, we can test other techniques and parameters and observe if they have any positive impact. Each parameter or method was tested separately.

It would make sense to start with the depth and width of the network, since those parameters shape the whole network. Starting with 64 neurons wide layers and trying to add up to six layers, it showed that more than three layers did not have any significant effect on the performance and unexpectedly adding more layers even had a negative impact. Table 4.1 shows the performance progression from increasing the depth of the network.

Depth	Width	Performance compared to optimum
1	64	52.3%
2	64	55.4%
3	64	56.1%
4	64	54.4%
5	64	54.6%
6	64	53.9%

Table 4.1: Performance improvement from increasing the depth of the mode

With three layers being the most promising depth so far, the next step was to increase the number of neurons in the layers. Increasing width did help to some degree, but the impact got smaller as the width increased. It also impacted the running time, so it was decided to have the model with three layers and 640 nodes in each layer (except input/output layers). Results were, however, still rather low at 58%. Table 4.2 shows the performance progression from increasing the width of the network.

Depth	Width	Performance compared to optimum
3	64	56.1%
3	120	55.8%
3	240	57.2%
3	360	56.8%
3	480	57.5%
3	640	58.0%

Table 4.2: Performance improvement from increasing the width of the model

Depth and width of the network affect the network's ability to capture more complex patterns, so it is expected that these parameters will play a larger role given larger and more diverse training datasets.

The Adam optimizer allows for the use of weight decay hyperparameter, which can help to keep weight values relatively small and decrease chances of the model memorizing the answers, also called overfitting. The default version of the Adam does not use weight decay, however, setting it to the commonly used values such as 0.1, 0.01 and 0.001 affected the results negatively, even during later training attempts with larger dataset.

Adding skip connections to the architecture creates an alternative way for information to flow through the network. Skip connections help the model to retain information from previous layers. No significant effect was noticed with only tenths of a percent improvement in performance, possibly due to the network having too few layers. Still, it was decided to keep the skip connections since they did not have any negavite effect and could be helpful later when we increase the size of the dataset or attempt to train the model on the more complex graphs.

Augmenting node features was another technique that was tested. The four features in the 4.1 figure stand for:

- Degree - neighbour count
- Relative difference - weight relative to the sum of neighbours. For each vertex v:

$$v.weight \div (\sum_{n \in N(v)} n.weight)$$

- Sum - sum of the neighbouring node weights. For each vertex v:

$$(\sum_{n \in N(v)} n.weight)$$

- Weight difference - difference between the nodes weight and the sum of the neighbours weigths. For each vertex v:

$$v.weight - (\sum_{n \in N(v)} n.weight)$$

Each node feature was added and tested separately on the initial model with two, 64 neuron layers, to see if they have any benefits by themselves. Then they were added one by one to see the cumulative effect. As a result, all the features were included in the preprocessing since they can, for the most part, be calculated simultaneously. All the features have shown to have a significant positive effect both on their own, and together.

Resulting in a final performance of 92% compared to greedy and 87% compared to the optimal solution:

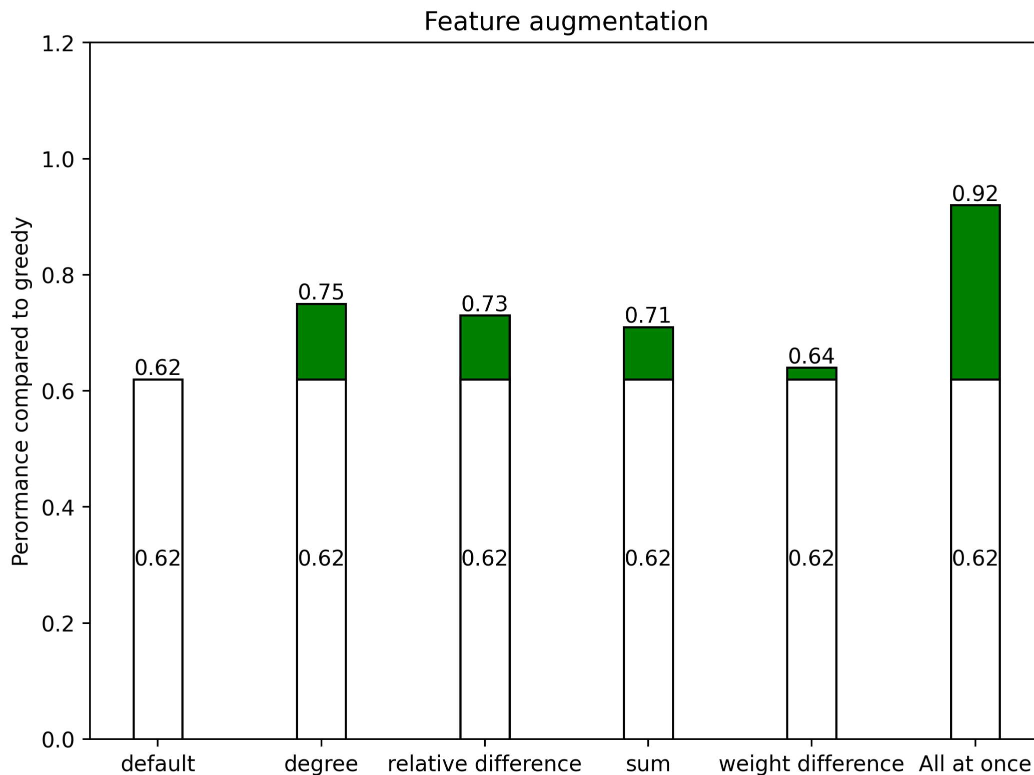

Figure 4.1: MNIST trained model. Average performance on 100 MNIST graphs

The figure 4.1 shows each feature's impact on performance. The default white column demonstrates the model's performance without any additional features. The green colored columns visualize the positive effect of adding a feature, while the last column represents the final result when all the features are added simultaneously. We can observe that every feature has a positive effect on the performance and gives the largest improvement so far compared to the other experiments.

With the features added to the nodes, we proceed to experiment with the aggregation functions for the message passing. In the table 4.3 one can see how the performance changes depending on the aggregation function used.

With fine-tuning of the models' hyperparameters and architecture, and augmenting node features having the most effect on the performance, results were moving closer to being reasonable. At this point, a full training set can be used to see more realistic results. The results were relatively promising, but further testing showed that, unfortunately,

Aggregation function	Performance compared to greedy
add	92.0%
sum	89.5%
mean	90.2%
min	87.0%
max	94.4%

Table 4.3: Aggregation function performance

line graph transformation on large and dense enough graphs turned out to grow out of proportions and take too much time to process. Therefore, another architecture was considered.

4.2 Edge prediction model

Instead of converting edges into nodes, a model can give predictions on the edges directly. With this approach, there is less preprocessing needed and without line graph convertion the input does not grow in size, but the model now needs to have a layer for the edge prediction part of the model. The purpose of this layer is to classify the edges based on the embeddings produced by the initial model, this is described in more detail in the Architecture section. A simple linear layer with 320 neurons was chosen as a starting point for the edge prediction.

After training both the edge classification model and the line graph model on the 55000 MNIST graphs with the best hyperparameters discovered in previous section, the following results compared to the greedy algorithm were recorded: 101% for edge classification and 103% for line graph model.

The figure 4.2 depicts how MNIST trained line graph and edge prediction model's compare in terms of performance on unseen MNIST graphs. The orange section of the bar represents the portion of the total weight that stemmed from the greedy algorithm solving the unsolved remains of the graphs.

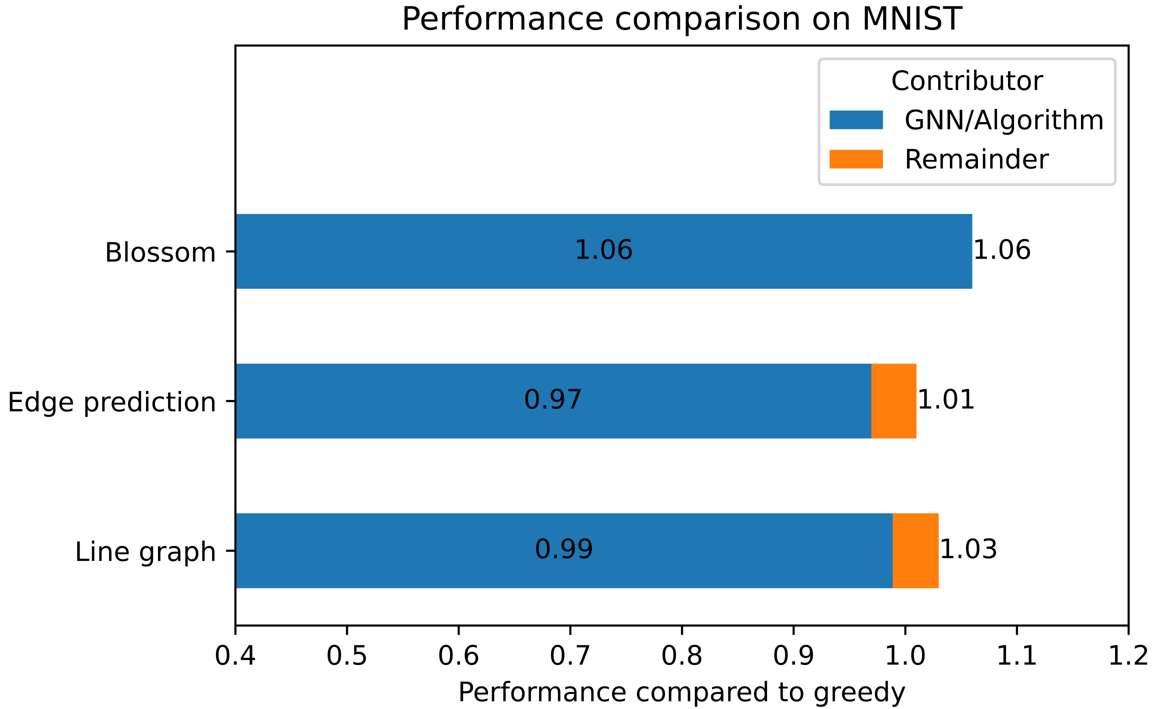

Figure 4.2: MNIST performance comparison

Edge classification showed a slightly worse results on average than the line graph. In theory, this can be due to the line graph containing more structural information in it compared to the original one. Regardless of the slightly worse result, the edge classification approach was still favorable due to the large time consumption of the line graph conversion. Overall, the results themselves are rather promising. Both models manage to beat the greedy algorithm even when the greedy result is close to the optimal. However, running time was still a problem. Due to the small size of MNIST graphs, finding an optimal solution using the Blossom algorithm takes less time than using our GNN model, because of the overhead computations needed for the neural network during the prediction phase, but the model should be able to catch up time-wise when given larger graphs. There is still the question of whether training on small graphs can help with larger graphs. The next step is to test the current model on a different graph - Cage10 from SuiteSparse. Cage10 is a graph with 11000 nodes and 100 000 edges. The results can be seen in figure 4.3:

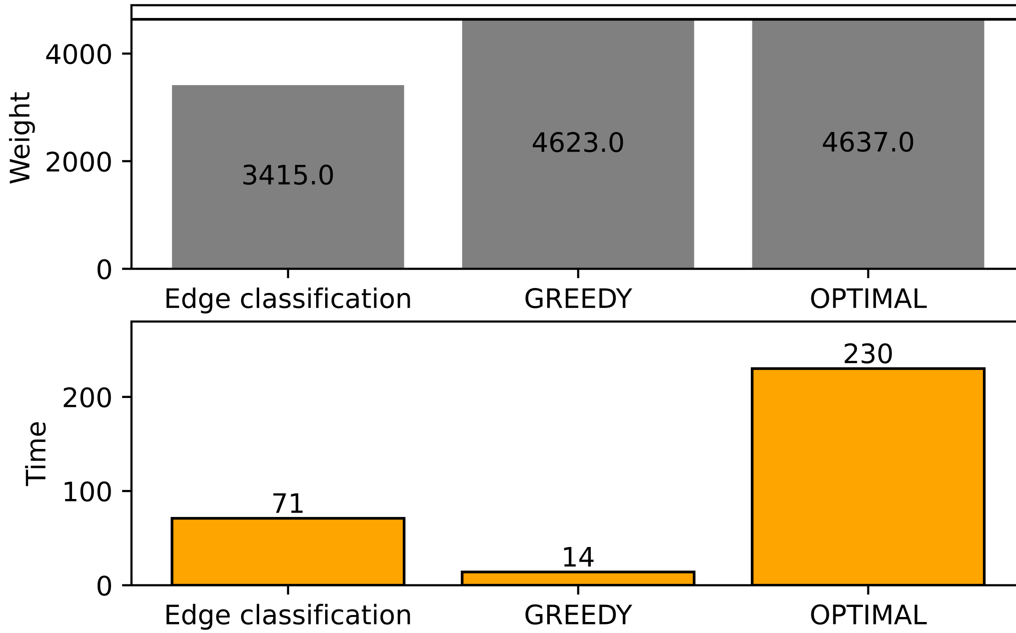

Figure 4.3: MNIST trained model performance on cage10 graph

As seen in figure 4.3 the GNN model manages to beat the optimal solver in terms of time, but it is noticeably worse in terms of total weight. Poor performance can be caused by the current training dataset. One of the main interests of the study was to see if the GNN could be used on graphs larger than the ones it was trained on. It could be that the model fails to learn what is needed for larger graphs. Another reason could be that MNIST graphs have similar structures without enough variety to cover other graph types. One, of course, cannot include all the possible graphs in the training set, but the more data and variety the model gets during training the better. Therefore, another dataset was tested for training. A custom dataset consisting of randomly picked graphs from the SuiteSparse database. This custom dataset consists of 1000+ graphs with varying sizes and structures, between 100 and 10000 nodes.

Training the same model on the new dataset showed that the model struggled to learn the patterns. The flat information loss during training indicated that the model experienced difficulties learning the more diverse collection of graphs. Adding two more layers to the classifier module helped with the stagnating learning curve, but the number of epochs had to be increased up to 500, due to the learning curve slowing down significantly in comparison to the previous dataset. Further adding more layers or neurons did not give any noticeable difference in performance. Figure 4.4 compares the performance of the new model trained using a custom dataset against the same model trained on

MNIST graphs only. Figure 4.5 shows the performance and the running time in seconds for GNN, greedy and blossom's algorithms.

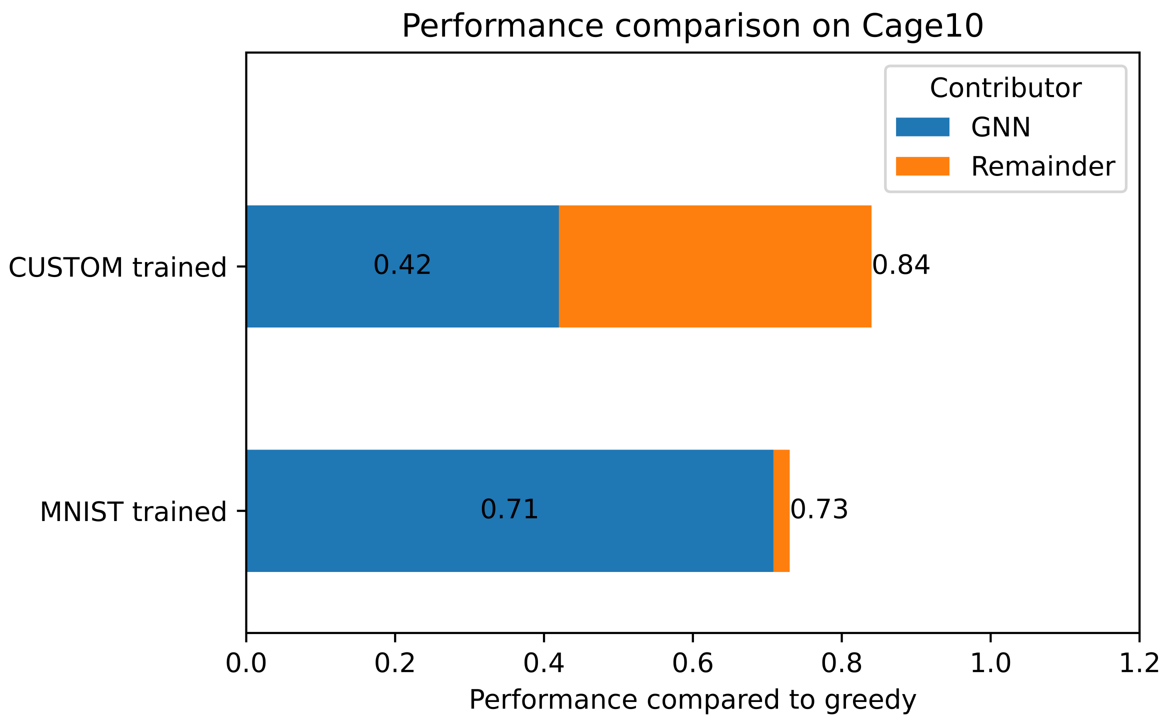

Figure 4.4: MNIST vs CUSTOM dataset training model performance on cage10

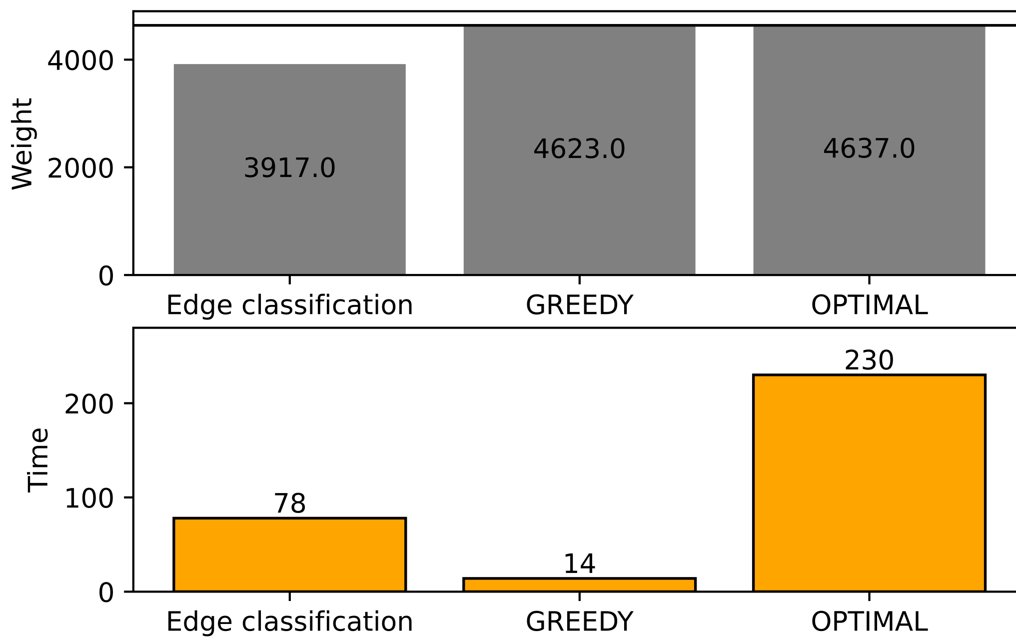

Figure 4.5: CUSTOM trained model performance on cage10 graph

The custom-trained model does perform better than the MNIST-trained model on cage-10 graph. However, it is not necessarily the model's accuracy itself that is the reason for improvement. The edge prediction model seems to be rather indecisive on which nodes to match and half of the graph at the end is left unmatched, as opposed to the MNIST-trained model matching almost the whole graph. The large remainder is not necessarily a problem in itself, since one can use the model to only match the most important nodes with the highest probabilities and leave the rest to the greedy algorithm to achieve better results. However, the custom-trained model does not leave a large remainder in every case as figure 4.6 demonstrates.

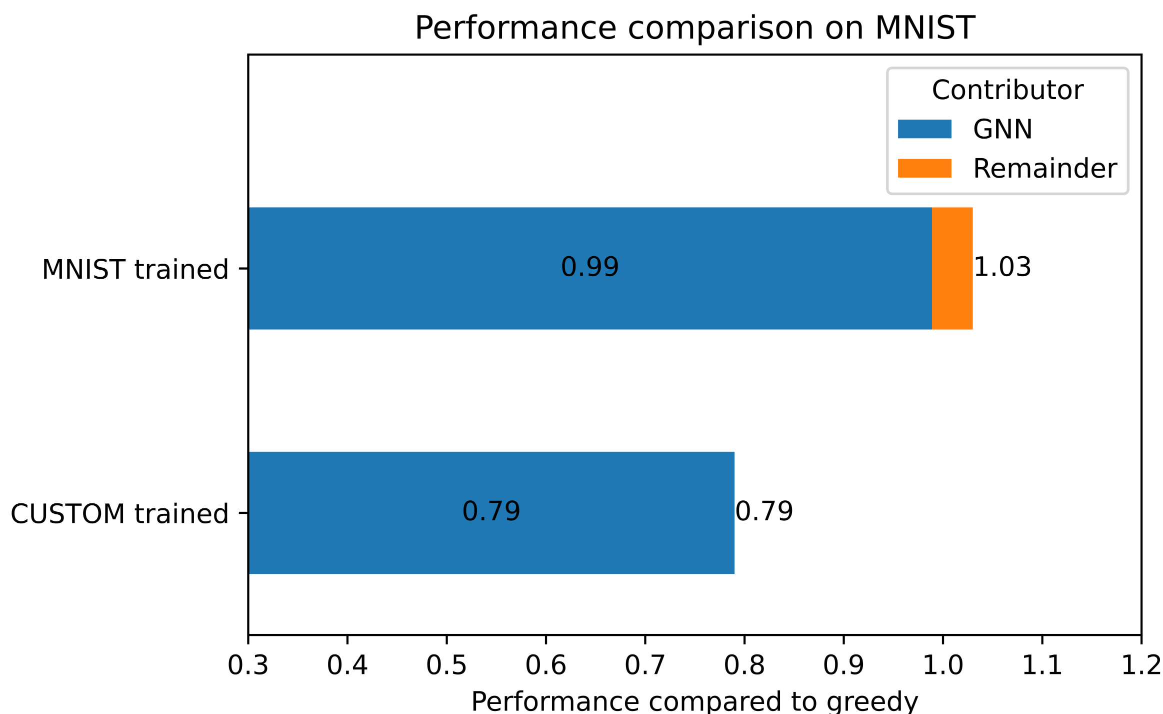

Figure 4.6: MNIST vs CUSTOM dataset training model performance on MNIST graphs

Although the custom-trained model leaves no remainder for MNIST graphs, it performes noticeably worse than the model trained on exclusively MNIST graphs. The custom dataset had fewer but larger graphs than the ones in the MNIST dataset. Performance on MNIST graphs could be improved by adding a subset of MNIST graphs to the custom dataset, but that might not help with graphs from other datasets.

The current results indicate that our model might fit better for use on a narrow family of graphs that belong to a common dataset. Although, there are two more modifications that might improve the current model:

- Cases that leave large remainders lead to the idea of adjusting the matching threshold to focus the model on only searching for the most important edges.
- A common practice in algorithms is to add reduction rules to reduce the size of the graph without affecting the result, one can add a reduction preprocessing step to assist the model.

4.2.1 Reduction

Reduction rules in algorithms are used to cut down the graph's size by removing the parts that are either guaranteed to be in the solution or the opposite. In the case of MWM, it is the edges that must be included in the solution regardless. Ideally, the neural network should be able to pick up such edges by itself, but it is still worth to test if there is any positive effect as well as to check if the model manages to grasp such rules by itself. The reduction rule itself is rather simple. If an edge that connects two nodes has larger weight than the sum of weights of the largest outgoing edges from each node, there is no reason not to pick it, since it is impossible to get a larger weight by matching these two nodes with other neighbours. Figure 4.7 serves as an example, where it is clear that node one should be matched with node five, if not, one has to match both one and five to other nodes. Second-best matchings for both node one and five have $weight = 3$, giving a total weight of only six.

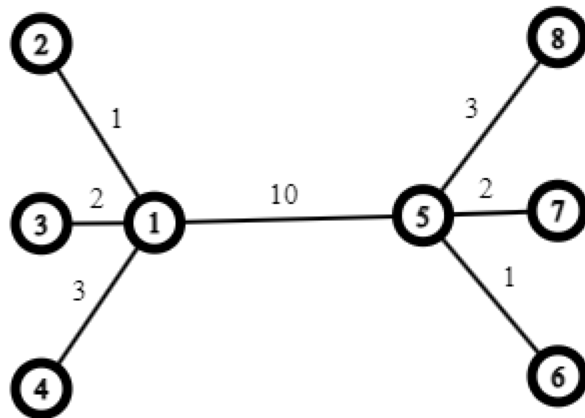

Figure 4.7: Reduction example

A couple of graphs from the custom dataset were used to test if the reduction helps and whether the model could identify reducible edges on its own. Some graphs had zero reducible edges, but in other cases such edges accounted for up to 5% of the total number

of edges. The model, however, did not seem to identify those edges in the graphs that contained them, and adding reduction as a preprocessing step did improve the results by between 2% and 3%. Since the reduction is based on the heaviest edges of the nodes, it is reasonable to add these node features to the model to potentialy help the model with finding the pattern for reduction.

The new model was trained with two additional node features: largest edge weight and second-largest edge weight for each node. Adding such features could in theory have helped the model to find reducible nodes, but after training the new model with the new node features, the results did not improve. The reducible edges were still left unidentified by the model.

4.2.2 Matching threshold

Adjusting the threshold for picking the edges is another way to use the model. The model was initially set to consider all the edges that have a larger than 50% chance of being in the solution. This matching threshold can be adjusted to, for example, only pick the edges with 90% probability of being in the solution. This can help to focus the model on the edges that have a large impact on the total result, but might be otherwise ignored by a greedy algorithm. Additionally, it can be worth testing the removal of edges with low probabilities to see if the model can find edges that, if picked, inhibit the overall result.

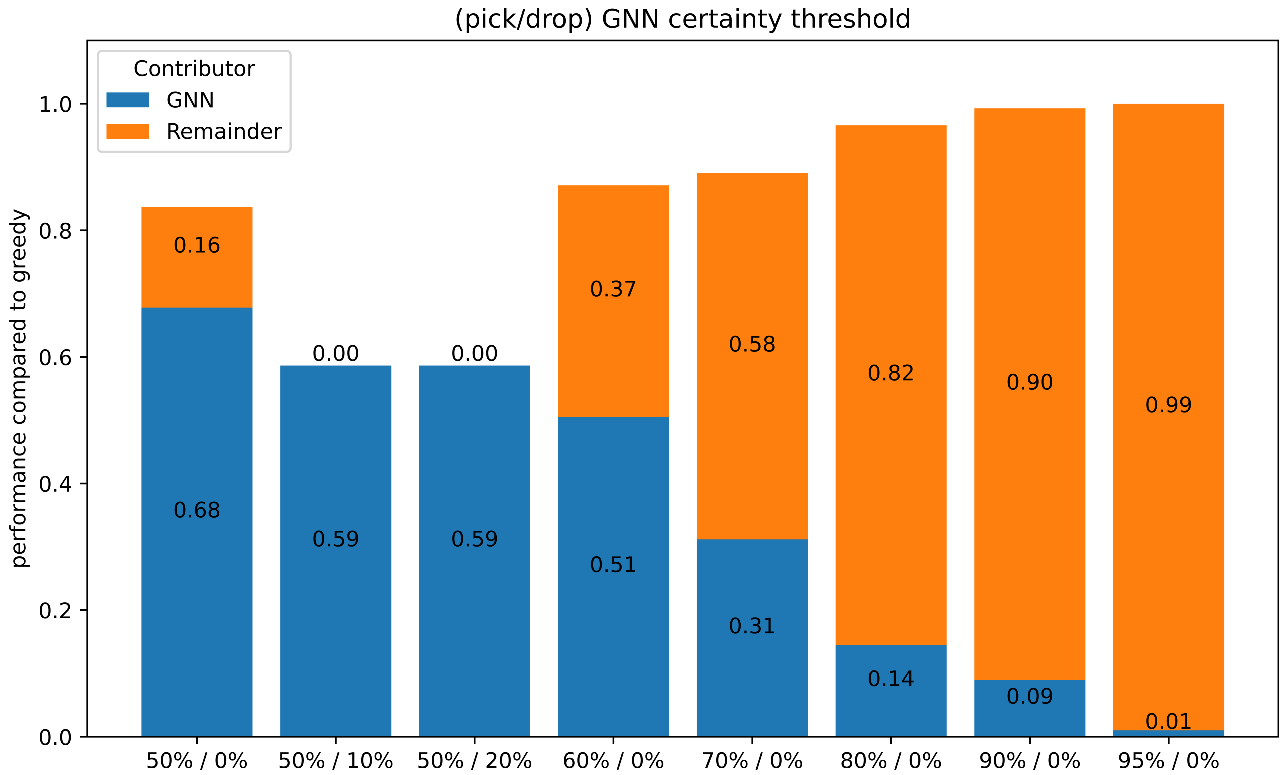

Figure 4.8: Model threshold test on cage8 graph

Figure 4.8 shows how different thresholds affect the performance of the model. The x-axis labels represent the thresholds for picking and dropping the edge respectively. For example, in the second column labeled "50%/10%", 50% refers to the lowest prediction certainty required for an edge to be in the solution and 10% refers to the edges with prediction certainty of being in the solution lower than 10% to be removed from the graph comepletely. The remainder is solved by the greedy algorithm.

The higher matching threshold does seem to improve the performance, but not enough to beat the greedy algorithm and most likely the increase is caused by the larger remainder being solved by th greedy algorithm due to fewer edges being above the threshold. Adding the drop threshold did not seem to work that well either. The model seems to assign very low scores to a lot of edges, which results in worse performance and no remainder for greedy algorithm to compensate. The reason for that is probably the weight class hyperparameter used during training, which makes the model prioritize less correctly finding nodes that should be ignored. The better approach could be to train the model specifically to find high value edges.

4.3 Final results

The final best model was tested on increasingly larger graphs from different sources. The final parameters of the model were:

1. Learning rate = 0.001
2. Epochs = 500
3. Mini-batch size = 1
4. Node embedding network depth and width = 3 layers, 640 neurons each
5. Edge classifier network depth and width = 3 layers, 320 neurons each
6. Class weights = 0.1 and 0.9 for dropped and picked edges respectively
7. Weight decay = 0
8. Match threshold = 70%
9. Added pre computed node features
    - Degree - how many neighbours a node has.
    - Sums of the weights.
    - Weights sum relative to the neighbours.
    - Weights sum difference compared to the neighbours.
10. Agreggation function = Max

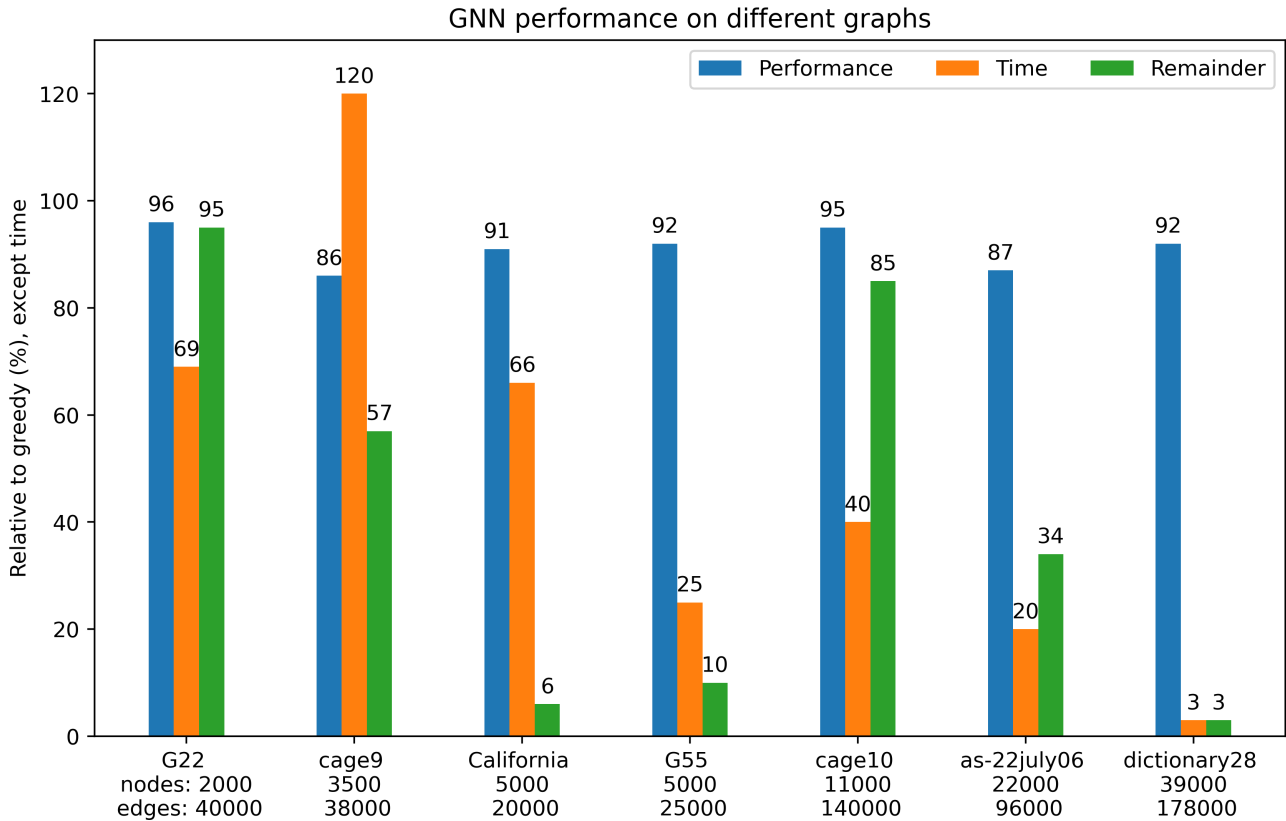

Figure 4.9: Final model performance

Figure 4.9 shows the performance of our final edge prediction model on the selection of increasingly larger graphs from different datasets. The blue colored columns stand for the models' performance on the graphs compared to the greedy algorithm. The green columns show the performance gained from solving the remainder of the graph with the greedy algorithm after the model could not find more edges. The orange column demonstrates the running time compared to the Blossom algorithm's running time.

Let's summarize the end results based on the main criteria.

- Performance: The total weight of the model's output seems to be rather stable at around 90% of what a greedy algorithm produces, regardless of the size of the graph and the remainder. It seems to be rather challenging for the model to beat the greedy approach when, in most cases, it makes up at least 85% of the total weight possible.

- Running time: An exception in the sense that the time is shown in comparison to the time it takes to find the optimal solution. The reason for that is the fact that the model is always slower than the greedy algorithm. As expected, the benefit of using the GNN can be seen as the size of the graphs increases. Smaller graphs take more time for the model to solve due to the overhead computations needed, but as the graphs grow the running time can get as small as 3% of the Blossom algorithm.
- Remainder: In some cases, the reason for the total weight being close to the greedy is the fact that the remainder makes up a large part of the graph, but it is not a consistent trend, and neither does it depend on the size of the graph. There are cases with equally good results where the remainder is below 10% of the total graph. The fact that some graphs leave such a big remainder indicates that the training dataset is not good enough and the model is not trained well enough to handle such graphs.

4.3.1 Weakness of the greedy algorithm

On average, a greedy algorithm seems to show good results, but it is not hard to make a graph that abuses the greedy approach and results in poor performance.

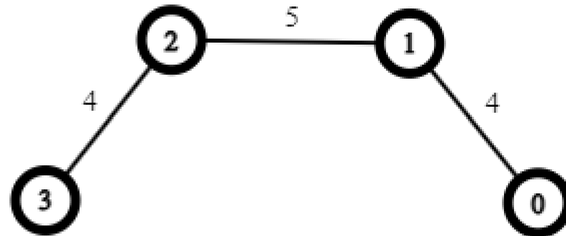

Figure 4.10: Good case for GNN

Figure 4.10 shows a simple case where the greedy algorithm results in total $weight = 5$. While our GNN manages to get the optimal $weight = 8$. Graphs that we found on SuiteSparse and graphs with random weights result in the greedy algorithm achieving at least 85% of the optimal result in the vast majority cases. Finding graphs that put greedy algorithm in disadvantage showed to be challenging, therefore, we modified the graphs used for testing using the principle showed in figure 4.10.

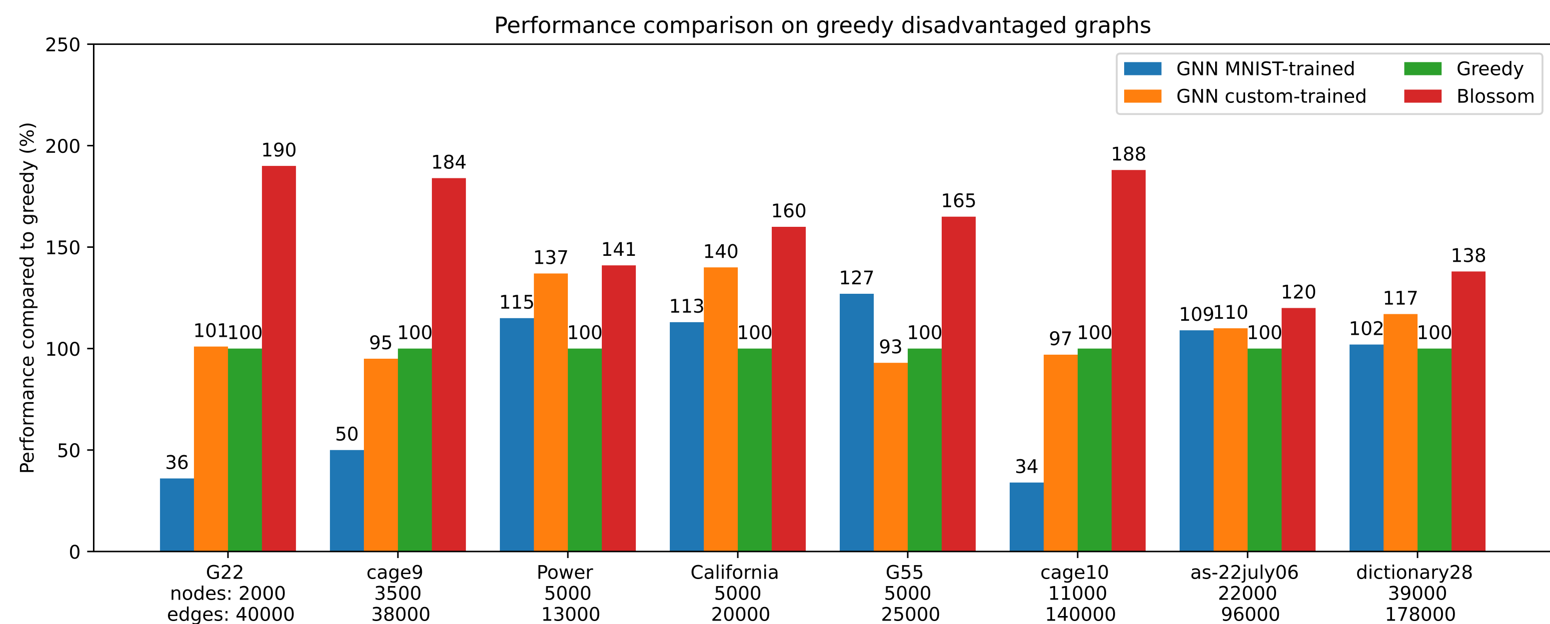

Figure 4.11: Greedy disadvantaged graphs

Figure 4.11 presents a performance comparison among the MNIST-trained model, custom-trained model, greedy algorithm, and Blossom algorithm when applied to graphs intentionally designed to challenge the greedy algorithm. For some of these modified graphs, our models outperform the greedy algorithm, a contrast to the results observed with the original graphs. However, for graphs from *cage* and G datasets, the performance is either worse than greedy or better than greedy, but significantly worse compared to the optimal solution. This suggests that our GNN models may be more effective for certain types of graphs rather than universally applicable across all graph categories.

Chapter 5

Conclusion

The model trained on graphs collected from multiple datasets showed some level of "understanding" of the task at hand and in special cases managed to beat the greedy algorithm. The best results achieved by the approach presented in this work showed that GNNs are capable of beating the greedy algorithm, but on average the performance was neither good nor consistent enough. The model on average achieved 90% of what the greedy algorithm did, except for the rare cases where the greedy algorithm was further away from the optimal result. One of the reasons why our GNN could not beat the greedy algorithm was due to the performance of the greedy algorithm being in most cases more than 90% of the optimal result.

Applying models to the graphs intentionally made to challenge the greedy algorithm resulted in better performance, where the models outperform the greedy algorithm in some cases, yet fail at other subsets of graphs. Varying performance on different sets of graphs might be due to the lack of variety within the datasets used for training and the insufficient complexity of the model. Since the input graph can be virtually of any shape and form, it might be hard to cover all the possible cases. The use of our GNN model might be better suited for the families of graphs that on average give worse results for greedy algorithm. However, such graphs were hard to find.

The models trained to specifically handle MNIST dataset graphs managed to outperform the greedy algorithm by 3% with the line graph implementation and 1% with the edge prediction, while the optimal result was on average only 6% better than greedy. However, the graphs were so small that the Blossom algorithm was in fact faster, although the GNN's running time confidently outperforms Blossom algorithm as the graphs' sizes

grow. The experiments on MNIST graphs strengthen the theory that a better use for such a model could be in cases where all the graphs belong to some subclass of graphs which narrows down the variety.

5.1 Future work

The fact that our GNN model developed in this work underperformed does not necessarily mean that the GNNs are in general unfit for MWM problem. There are several potential improvements that can be made.Various architectures and methods can be tested alongside larger datasets. Increasing the number of layers and neurons can enhance a model's ability to recognize complex patterns, albeit at the expense of longer training and prediction times. However, for this specific architecture, adding more layers had little to no impact.

Another problem might be rooted in the supervised approach. Training a model based on the optimal solution has its pitfalls. The optimal solution can be completely different if even one edge is classified incorrectly. A semi-supervised or an unsupervised approaches are good potential candidates where precomputing the optimal solution would not be needed.

Glossary

Artificial Intelligence Artificial Intelligence is a field of study regarding intelligence simulated by computers. Where intelligence is meant in context of human intelligence.

Git Git and GitHub is a Version Control System (VCS) for tracking changes in computer files and coordinating work on those files among multiple people.

Machine Learning Machine Learning studies algorithms that learn general patterns and make predictions based on some form of input. It is one form of Artificial Intelligence.

Neural Network Neural Network is one of many technologies used inMachine Learning.

List of Acronyms and Abbreviations

CNN Convolutional Neural Network.

CO Combinatorial Optimization.

GCN Graph Convolution Network.

GNN Graph Neural Network.

MIS Maximal Independent Set.

MWIS Maximum Weighted Independent Set.

MWM Maximum Weighted Matching.

PyG Pytorch Geometric.

VCS Version Control System.

Bibliography

[1] Lada A Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2003. ISSN 0378-8733. doi: https://doi.org/10.1016/S0378-8733(03)00009-1.
URL: https://www.sciencedirect.com/science/article/pii/S0378873303000091.

[2] Maria Chiara Angelini and Federico Ricci-Tersenghi. Modern graph neural networks do worse than classical greedy algorithms in solving combinatorial optimization problems like maximum independent set. *Nature Machine Intelligence*, 5(1):29–31, December 2022. ISSN 2522-5839. doi: 10.1038/s42256-022-00589-y.
URL: http://dx.doi.org/10.1038/s42256-022-00589-y.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016.

[4] Lorenzo Brusca, Lars C. P. M. Quaedvlieg, Stratis Skoulakis, Grigorios G Chrysos, and Volkan Cevher. Maximum independent set: Self-training through dynamic programming, 2023.

[5] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks, 2022.

[6] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17: 449–467, 1965. doi: 10.4153/CJM-1965-045-4.

[7] M. Fujii, T. Kasami, and K. Ninomiya. Optimal sequencing of two equivalent processors. *SIAM Journal on Applied Mathematics*, 17(4):784–789, 1969. doi: https://doi.org/10.1137/0117070.
URL: https://doi.org/10.1137/0117070.

[8] Ghosh and Reilly. Credit card fraud detection with a neural-network. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 3, pages 621–630, 1994. doi: 10.1109/HICSS.1994.323314.

[9] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[10] Frank L. Hitchcock. The distribution of a product from several sources to numerous localities. *Journal of Mathematics and Physics*, 20(1-4):224–230, 1941. doi: https://doi.org/10.1002/sapm1941201224.
URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/sapm1941201224`.

[11] Kh Tohidul Islam, Ghulam Mujtaba, Ram Gopal Raj, and Henry Friday Nweke. Handwritten digits recognition with artificial neural network. In *2017 International Conference on Engineering Technology and Technopreneurship (ICE2T)*, pages 1–4, 2017. doi: 10.1109/ICE2T.2017.8215993.

[12] Ron Karjian. Weighted maximum matching in general graphs, 2023.
URL: `https://www.techtarget.com/whatis/A-Timeline-of-Machine-Learning-History`.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[14] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.
URL: `http://arxiv.org/abs/1609.02907`.

[15] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[16] Wentong Liao, Kai Hu, Michael Ying Yang, and Bodo Rosenhahn. Text to image generation with semantic-spatial aware gan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 18187–18196, June 2022.

[17] NIST. undirected graph definition. `https://xlinux.nist.gov/dads/HTML/undirectedGraph.html`, 2007. [Online; accessed 04-May-2024].

[18] Amirhossein Nouranizadeh, Mohammadjavad Matinkia, Mohammad Rahmati, and Reza Safabakhsh. Maximum entropy weighted independent set pooling for graph neural networks. *CoRR*, abs/2107.01410, 2021.
URL: `https://arxiv.org/abs/2107.01410`.

[19] PyG Team. Pyg documentation. `https://pytorch-geometric.readthedocs.io/en/latest/`, 2024. [Online; accessed 17-April-2024].

[20] Rajat Raina, Anand Madhavan, and Andrew Ng. Large-scale deep unsupervised learning using graphics processors. volume 382, page 110, 06 2009. doi: 10.1145/1553374.1553486.

[21] Arvind Ramachandran, Nicholas Frosst, and Geoffrey E Hinton. Protein interface prediction using graph convolutional networks. In *Advances in Neural Information Processing Systems*, pages 11128–11137. 2019.

[22] Partha Pratim Ray. Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. *Internet of Things and Cyber-Physical Systems*, 3:121–154, 2023. ISSN 2667-3452. doi: https://doi.org/10.1016/j.iotcps.2023.04.003.
URL: `https://www.sciencedirect.com/science/article/pii/S266734522300024X`.

[23] Alvin E. Roth, Tayfun Sönmez, and M. Utku Ünver. Pairwise kidney exchange. *Journal of Economic Theory*, 125(2):151–188, 2005. ISSN 0022-0531. doi: https://doi.org/10.1016/j.jet.2005.04.004.
URL: `https://www.sciencedirect.com/science/article/pii/S0022053105001055`.

[24] Martin J. A. Schuetz, J. Kyle Brubaker, and Helmut G. Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, 4(4):367–377, April 2022. ISSN 2522-5839. doi: 10.1038/s42256-022-00468-6.
URL: `http://dx.doi.org/10.1038/s42256-022-00468-6`.

[25] Madhusmita Swain, Sanjit Kumar Dash, Sweta Dash, and Ayeskanta Mohapatra. An approach for iris plant classification using neural network. *International Journal on Soft Computing*, 3(1):79, 2012.

[26] Boukaye Boubacar Traore, Bernard Kamsu-Foguem, and Fana Tangara. Deep convolution neural network for image recognition. *Ecological Informatics*, 48:257–268, 2018. ISSN 1574-9541. doi: https://doi.org/10.1016/j.ecoinf.2018.10.002.
URL: `https://www.sciencedirect.com/science/article/pii/S1574954118302140`.

[27] Joris van Rantwijk. Weighted maximum matching in general graphs, 2008.
URL: `https://github.com/ageneau/blossom/blob/master/python/mwmatching.py`.

[28] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[29] Wikipedia contributors. Maximum weight matching — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Maximum_weight_matching&oldid=1169853595`, 2023. [Online; accessed 17-April-2024].

[30] Bohao Wu and Lingli Li. Solving maximum weighted matching on large graphs with deep reinforcement learning. *Information Sciences*, 614:400–415, 2022. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2022.10.021.
URL: `https://www.sciencedirect.com/science/article/pii/S0020025522011410`.

[31] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, IJCAI-2018. International Joint Conferences on Artificial Intelligence Organization, July 2018. doi: 10.24963/ijcai.2018/505.
URL: `http://dx.doi.org/10.24963/ijcai.2018/505`.

Appendix A

Code examples

Simple GCN in Python example:

```
1  class MyGCN(torch.nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.conv1 = GCNConv(5, 640) # 5 features per each node,
               ↪ initialy 1
5          self.conv2 = GCNConv(640, 640)
6          self.lin = Linear(640, 2) # 2 classes
7
8      def forward(self, data):
9          x, edge_index, edge_weight = data.x, data.edge_index,
               ↪ data.edge_weight
10
11         x = self.conv1(x, edge_index, edge_weight)
12         identity = F.relu(x)
13
14         x = self.conv2(x, edge_index, edge_weight)
15         x = F.relu(x)
16
17         return F.log_softmax(x, dim=1)
```

Edge classification example in Python

```
1
2  class MyGCNEdge(torch.nn.Module):
3      def __init__(self):
4          super().__init__()
5          self.conv1 = GCNConv(7, 640)
6          self.conv2 = GCNConv(640, 640)
7          self.encode = Linear(640, 80) # 80 output features per node
8
9      def forward(self, data):
10         x, edge_index, edge_weight = data.x, data.edge_index,
               ↪ data.edge_weight
11
12         x = self.conv1(x, edge_index, edge_weight)
13         identity = F.relu(x)
14
15         x = self.conv2(x, edge_index, edge_weight)
16         x = F.relu(x)
17
18         x = self.encode(x)
19         return x
20
```

```
21 class EdgeClassifier(torch.nn.Module):
22     def __init__(self):
23         super().__init__()
24         self.lin1 = Linear(160, 320) # 80 * 2 features since input is
               ↪ 2 nodes.
25         self.lin2 = Linear(320, 2)
26 
27     def encodeEdges(self, nodeEncode, graph):
28         x_src, x_dst = nodeEncode[graph.edge_index[0]],
               ↪ nodeEncode[graph.edge_index[1]]
29         edgeEncode = torch.cat([x_src, x_dst], dim=-1)
30         return edgeEncode
31 
32     def forward(self, edgeEncode):
33         x = self.lin1(edgeEncode)
34         x = F.relu(x)
35 
36         x = self.lin2(x)
37         return F.log_softmax(x, dim=1)
```