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

March, 2024

Abstract

In this work we tried to 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 greedy algorithm on average performed very well compared to the optimal solution. However, the neural network did show potential at solving specific gaphs that abuse the way greedy algorithm works. The results also do not imply that graph neural networks in general cannot beat greedy algorithm, but rather suggest that different approach or further improvement are needed.

Acknowledgements

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

Nikita Zaicev
Sunday 31st March, 2024

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

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

1.1 Goal

Machine Learning and Neural Networks have shown to be extremely potent and versatile in solving vast variety of problems across different fields. One research field that has been popular and challenging in the last few years is Combinatorial Optimization (CO). CO includes problems such as Maximal Indepndent Set (MIS) and Maximum Weighted Matching (MWM). Such problems can be viewed in the context of graphs. Graph Neural Network (GNN) is a subclass of Neural Networks designed specifically for solving problems related to graphs, but there are some 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 Reason

Previous section describes "what" is this work about. Now lets briefly discuss "why" do this in the first place. The idea is rather simple. There are 2 types of algorithms in general: exact and approximate. The exact algorithms for MWM already exist, but the

down side of such algorithms is their speed. The slower nature of exact algorithms is one of the reasons why approximation algorithms are used in some cases. The hope is that GNN can potantialy squeeze in between the approximation and exact algorithms in terms of both time and accuracy.

In short, the reason 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 will briefly touch the history, applications and impact of machine learning and go through core concepts and ideas behind machine learning and neural networks. Relevant information about CO and relevant algorithms will also be included here.

2. Methodology and Data:

   This Chapter will focus on how the research was done and also explain core concepts of Neural Networks and the general procedure of training a Neural Network. Then the specifics of this case will be discussed together with the main challenges. The progress made step by step and the reasoning behind changes and choices made along the way will be shown. The chapter will also analyze the data used for training the model and evaluating results along with justification for the chosen data.

3. Results:

   This Chapter will focus on temporary results produced during the reseach and explain how these results affected the further decisions. Finally the chapter will be concluded by analyzing the final results and comparing them with the expectations.

4. Conclusion:

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

1.3.1 Git

The whole project can be seen here: `https://github.com/nikitazaicev/Master`

Chapter 2

Background

This chapter explains all the relevant concepts, technologies and problems relevant to this project.

2.1 Neural Networks and Machine Learning

Machine learning in computer science is a broad term for all the algorithms that use statistical methods to learn patterns and make predictions. Some of the earliest algorithms were invented already in the 1940s [5], but it is notably in recent times, with increasing computational power of the computers and GPU's, that machine learning and especially neural networks have shown impressive results in a variety of fields such as recognition and generation of images, videos, sound and text.

Machine learning can be further split into supervised and unsupervised learning. Supervised learning means the model is given correct answers it can learn from by trial and error and then apply gained knowledge on the unseen data. The other option is unsupervised learning where the model usually has some formula it can use to calculate how far it is from the set goal. In this case we will be trying supervised learning where we solve MWM problems and use them to train the model.

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 in 3 types: an input layer that reads the given data, one or more middle layers that proccess the data and output layer that give the final

answer. Each neuron in a layer is usually connected to all the neurons of the next layer, although it may depend on the architecture. Between connected neurons there are weights. When the neural network model recieves input, it is passed through the layers and gets recalculated using the weights. The result at the output layer is some altered numerical representation of the input that can, depending on the goal, be interpreted as some prediction, for example a probability. The weights of neural network can then be adjusted depending on the accuracy of the prediction. After the adjustment is done model can try to make a prediction again. This process can be repeated multiple times, until the model can make predictions well enough. This process is called training. An important thing to keep in mind is that a trained model needs to be tested on unseen data since it could have just memorized all the data it was trained on to make predictions. In machine learning one is interested in generalization, the model needs to find some patterns that are common for all the relevant data it can encounter.

2.2 Graph Neural Networks

GNN is a relatively young subclass of neural networks designed specifically for solving graph related problems and it has been a growing topic of research in the last couple of years.

Lorenzo Brusca and Lars C. P. M. Quaedvlieg et al. [2] showed a self-training GNN for MIS.

Schuetz et. al. made an unsupervised GNN [9] for solving MIS and Angelini and Ricci-Tersenghi [1] compared its performance to greedy algorithms and reported some problems with GNNs performance.

The reason MIS solving is mentioned so much here is because the problems are to some degree related, as algorithmic problems often are. There will be concrete examples in the Methodology and Data chapter. MIS is also more often a subject to research and researches on MWM are not that common. Bohao Wu and Lingli Li tried to solve MWM using deep reinforcement learning [11] and reported that "Experimental results show that L2M outperforms state-of-the-art algorithms."

GNNs are based on the same concepts as the neural networks, but with some modifications to better suit graph related problems. Opposite to the more standart neural networks, GNN makes a representation of each node based on its neighbours. Adding

more layers increases the depth of neighbourhood used. Meaning if a model has 2 layers the representation of one node would be calculated from nodes neighbours and the neighbours of the neighbours. This allows the GNN to utilize graph structure unlike a standart neural network.

2.3 Graphs

A short introduction to what a graph is and what terminology is used in this work.

Graph (from graph theory) - a data structure that contains items in form of nodes (also called vertices), where nodes can be connected. If two nodes are connected - they have an edge between them. Visualy graph are represented as dots (nodes) connected with lines (edges).

2.4 Maximum Weighted Matching and Combinatorial Optimization

Combinatorial optimization is a field of study that covers problems that require finding a subset or a combination from some set of elements, that fulfills certain requirements. An example of such problem would be Maximum Weighted Matching and Maximum Independent Set.

MWM problem asks to find a subset of node pairs in a weighted graph, such that sum of weights is maximized. The problem is relatively simple compared to many others is the field of CO and has optimal algorithms that run in polynomial time such as blossom algorithm [4]. Optimal algorithms however can still be rather slow when working with large datasets, and in some cases approximation algorithm may be more of a suitable solution if the exact answer is not critically important. The most simple example is a greedy algorithm that sorts all the edges by their weight and picks them in descending order.

The MIS asks to find a subset of non-adjecent nodes such that the total amount of nodes is as large as possible. There is a similar problem Maximum Weighted Indepndent Set (MWIS) which asks to find an independent set that gives the largest total weight sum. MWIS is partially relevant as we will later try a graph transformation that essentialy turn the original MWM problem into MWIS.

Chapter 3

Research Methodology and Data

This chapter focuses on describing which approaches were chosen for the task at hand and why, as well as shows how the work progressed.

To reiterate the current problem is to find out if GNN can compete at solving the MWM problem compared to other approximation methods such as greedy algorithm.

3.1 Challenges

It is common to think of a neural network as of a black box where you give some data to this box and it tries to predict the correct answer. In this case the data is a graph consisting of vertices connected by edges that have weights and the expected answer should be pairs of vertices that were matched together.

As mentioned the idea behind supervised learning is to give a model the correct answers so it can by trial and error learn from it. Theese answers are not included in the initial datasets so the optimal solutions needs to be calculated. To find the optimal solution Blossom algorithm implemented by Joris van Rantwijk in Python was used [10]. 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 for algorithmic problems for the large graphs can be time consuming, at the same time a model need as much data as possible to learn, leading to multiple large graphs consuming to much time combined. This poses

a question whether it is possible for the model to learn on small and medium sized graphs that are not as time consuming and transfer learned patterns to solve larger graphs.

Another important aspect that must be taken in to the consideration is that it is unlikely for the model to fully follow the restictions of the problem. Model might decide to match same node to 2 neighbors at the same time, which should not be allowed.

3.2 Main Idea

To summarize, given a weighted udirected graph, the Neural Network must predict a valid subset of edges or in other words pairs of vertices that maximize the total weight. The Neural Network will have a graph as an input and the otput will be the probabilities (0.00 - 1.00) of how likely an edge will be in the matching. To control that a solution is valid, instead of picking all the edges with the probability higher than 50%, the edges can be sorted by their probabilities and greedily picked in that order if they don't break the validity of the solution. This might sound illogical to use 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 Neural Network does make a difference, since the GNN can take multiple features in consideration when assigning probabilities. Some of these features come naturaly from the structure of the graph, and some we can manualy add during preprocessing.

Neural Networks have a lot of hyperparameters and methods that can be tuned to make a model better suited for the task. The ideal way of finding the best combination of the hyperparameters is to try as many combinations as possible. Train a model for each combination and choose the one with best performance. This is a time consuming procedure however and therefore in this task the exploration of hyperparameters was narrowed down to the ones that seemed most important and indicated positive impact during early experiments. For training, Adam optimizer was used [6]. Optimizer is used to adjust the weights of the network during training. Adam optimizer has several hyperparameters that can be usefull to achieve better results and is one of the popular optimizers to use.

During the experiments following hyperparameters were tested:

1. Learning rate - by how much the weights of the model should be corrected

2. Network depth - amount of layers
3. Network width - neurons in the layers
4. 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.
5. Weight decay - used to keep GNN weight values relatively small and decrease chances of the model memorizing the answers.
6. 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.
    - Sums of the weights.
    - 1st largest weight, 2nd largest weight.

3.3 Data

The model should be capable of solving any graph relevant to MWM problem. A relevant graph can be difined by following characterisitcs Ideally the model should be able to handle any kind of an udirected graph with weighted edges.

For training and experimenting with the GNN, a MNIST dataset was used [3]. 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 node. Theese features is what is used as the weights for the problem. Although the graphs in this dataset are relatively small they have some fitting qualities such and nodes having many neighbors creating more possible ways to match the nodes. Small size also makes it less time consuming to train the models and try different approaches as well as shows whether a GNN trained on smaller graphs can transfer its knowledge to larger graphs.

Only training and testing on MNIST graphs does not represent all the different graphs that can occur, therefore model had to be tested on other graphs that have different

structures and weight distributions. Model trained exclusively on MNIST dataset did not perform well on other graphs as one might ha expected. Therefore another dataset was made to cover larger variety of graphs. The dataset was a collection of random graphs from SuitSparse database that ranged between 100 and 10000 nodes. As described in more details in the Data preprocessing section, some of the found graphs did not have edge weights. In these cases random weights were generated.

Including truly large graphs in training caused problems with insufficient memory and too long training time, but final model was still tested on some larger graphs than used in training. A couple large graphs from SuiteSparse were chosen for testing performance on large graphs with 40000 nodes and higher.

Small handcrafted graphs were used to test if GNN can at least beat the cases specificaly made to abuse weaknesses of a greedy approach.

3.4 Architecture

During the experiments the architecture of the GNN model itself had mostly minor changes, but 2 rather different graph preproccessing steps were tried. In both cases we use Graph Convolution Network (GCN) layers (GCNConv) [7]. The GCNConv layer fits well for the purpose of this task because it makes use of the edge weights that are the crucial part for solving MWM.

Python example:

```
class MyGCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = GCNConv(5, 640) # 5 features per each node,
            ↪ initialy 1
        self.conv2 = GCNConv(640, 640)
        self.lin = Linear(640, 2) # 2 classes
```

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

3.4.1 Model pipeline

As mentioned before one cannot fully trust the model to satisfy the restrictions of the problem. What is meant by the restrictions of the probem is the fact that every node can only be matched once. This is controlled by sorting the probabilities 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 happen 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 pick the matches with high enough probability are used. As a starting point probability threshold is 50% and above. Two things are done 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 model this would essentialy just be a new graph and it may happen that 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, can be solved using standard greedy or any other approximation algorithm for that matter.

Passing the graph and it's consecutive remainders multiple times through the model can also be helpfull to deal with the case where one of the matches that was picked may play big role in how the rest of the solution will look like. Idealy to handle this case one might only pick one match at the time and pass the graph through the model again, but it will be too time consuming.

3.4.2 Line Graph Approach

Line graph approach was the first attempt at using a simple GNN, which later turned out to be too time consuming for larger graphs to be worth further expriements. It did however give some usefull insight as well as a showed to be a proof of concept. In the context of graphs 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.

Examaple of a graph and its line graph convertion

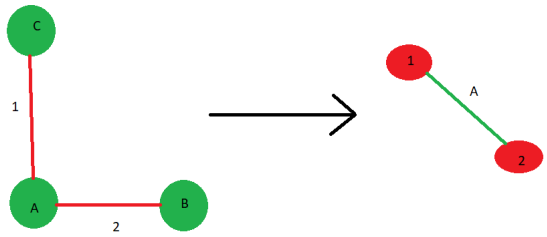

Figure 3.1: Original graph (left) and its line graph (right)

Converting to a line graph makes the architecture of the GNN model simpler. Instead of needing to ask model for each pair of nodes is they should be matched together, the line graph model now can output the probability of a node being in the matching directly, since the node now represents two connected nodes from original graph. This also however transforms the problem itself. From the perspective og the model it is now trying to solve MWIS. GNNs for MIS have been studied before. Nouranizadeh et. al. showed demonstrated a pooling method for solving MWIS [8]. Unfortunately the large graphs can explode in size when converted to line graphs and were too time consuming in such cases.

3.4.3 Edge Classification Approach

A more natural way of approaching this problem is to give the model original graphs instead of transforming them to line graphs. But the model itselv only returns some numerical representation on the nodes in the graph. To get the predictions for the edges one can add a classifier module to the existing GNN. This module's purpuse is to predict if a pair of two nodes will be in the matching. So 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. Resulting in the model output being probabilities of all the edges being in the matching.

Python example:

```
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.embed = Linear(1280, 80)
8
9 class EdgeClassifier(torch.nn.Module):
```

```
10      def __init__(self):
11          super().__init__()
12          self.lin1 = Linear(160, 320) # 80 * 2 features since input is
                ↪ 2 nodes.
13          self.lin2 = Linear(320, 2)
14
15      def embedEdges(self, nodeEmbed, graph):
16          x_src, x_dst = nodeEmbed[graph.edge_index[0]],
                ↪ nodeEmbed[graph.edge_index[1]]
17          edgeEmbed = torch.cat([x_src, x_dst], dim=-1)
18          return edgeEmbed
```

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 require some preproccessing of data before it can be used. Convertion to the line graph is one example of such preproccessing. One might also need to augment the initial data to improve model performance. Finally machine learning models can be vulnerable to large numerical swings in the data, such as edge weights being higher than expected.

Most important are the edge weights. To ensure we have enough graphs to train on and the graphs cover a variety of structures, some of the chosen graphs that lacked weights recieved a randomly generated weights. The random weights had even distribution in the range between 0 and 1. All the graphs with preexisting weights were also adjusted to be in that range by shifting the weights to positive range if any negative weights were detected and then dividing by the largest weight. The weights could have been in any range, but the important thing is to have consistent weights for all the graphs.

Another preprocessing step that was added is additional features for the nodes/vertices. This required different approaches for line graph and edge prediction approaches since in a line graph vertex represents an edge from the original graph. Line graph has weights assigned to the nodes themselves while using original graph nodes are assigned to edges.

One vertex feature does reamain in common: Degree of the nodes. Meaning how many neighbors each node has.

Line garphs node features:

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

$$Wrel = v.weight \div (\sum_{n=0}^{|neighbors|} neighbors[n].weight)$$

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

$$Wdiff = v.weight \div (\sum_{n=0}^{|neighbors|} neighbors[n].weight)$$

3. Sums of the neighbouring node weights.

$$(\sum_{n=0}^{|neighbors|} neighbors[n].weight)$$

Normal graph edge features. Here some adjustments are made since nodes do not have any weights assigned unlike line graph.

1. Sum of the weights of all the outgoing edges.

$$Wsum = (\sum_{i=0}^{|node.edges|} node.edges[i].weight)$$

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

$$Wrel = Wsum \div (\sum_{n=0}^{|neighbors|} neighbors[n].Wsum)$$

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

$$Wdiff = Wsum \div (\sum_{n=0}^{|neighbors|} neighbors[n].Wsum)$$

3.5 Result Validation

Before looking at the results it is important to decide how to evalueate 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 this includes model specific preproccessing such as adding additional features as well as finishing the potential remainder of the graph with greedy algorithm.

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

3. GNN portion - due to the way 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 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 of what 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 can often happen that during the project something goes wrong with the model and it can be not obvious. Therefore, one often does what is called sanity checks to see if model's inputs and outputs still make sense.

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 enough time and tested it on the same data. The model should be able to memorize theese graphs and solve the problem near perfect on them.

2. Comparisons to random matching were done. Before the training the model has random weights and essentialy will behave as if it is picking edges at random, but after training the model was performing better than a random matching.

3. The model was tested on a small hand crafted graphs that were made specificaly to put greedy algorithm in disadvantage. That at least showed that the model goes in the right direction even if it only manages to solve the easy cases.

3.6 Expected Results

3.6.1 Accuracy and total weight

There are not that many researches specifically for MWM, but problems like MIS that are relatively close to MWM can indicate similar results for this case as well. As discussed in the Background section, there are researches that show that GNNs are capable of solving CO problems and beating greedy algorithms, while other rather indicate that improvements are still needed for it to be worth using. Therefore it is hard to forecast any results based on previous work. Nothing stands in the way of being optimistic however, additionally to the fact that greedy algorithm is relatively simple and Neural Network should be able the recognise a more complex pattern it can use to achieve better results. It is absolutely not expected for the model to be able to find 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 absorved that for the majority of graphs greedy algorithm preforms rather well with above 80% of the optimal possible weight.

3.6.2 Time

GNN model is a heuristic solver and gives an approximate answer. Therefore model should be noticeably faster than 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 preproccessing required such as augmenting data with adittional features. Naturaly, time will also depend on the depth and the width of the network.

3.6.3 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 final result it would still be a usefull tool. A speculative estimation for the sake of having something for orientation will be 80%. That is the weight obtained from models prediction makes at least 80% of total weight.

3.6.4 Other observations

One improtant 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 is rather small. For the MNIST dataset and the majority of the graphs found on Suite Sparse Matrix Collection the greedy algorithm manages to reach 90+% of the optimal weight. Assigning random evenly distributed values to the edges weights also seem to give the same results. For the GNN this means that it will be rather difficult to beat the greedy algorithm since it gets very close to the optimal.

Chapter 4

Results

This chapter lists the temporary results from the experiments and thoughts on what it meant for the model, as well as the final results. All the experiments have been done on the same machine with: 11th Gen Intel(R) Core(TM) i7-11700K 3.60GHz 8-core CPU, 16 GB RAM and NVIDIA RTX 3080Ti graphics card.

4.1 Progress

In this section we will go throught the whole proccess step by step, from first iterations of the very simple model to more fine tuned models and other methods that were tested.

4.1.1 Simple line graph model

First step was to try the simplest model with most of the hyperparameters set to default and train it on a smaller scale. First model had 2 layers and 64 neurons each. Initially due to majority of nodes being not included in the matching, the model learned to set all nodes to be dropped and still get a rather high accuracy score. This was resolved by adding class weights as a training parameter. Class weights tell the model how important each class is. By assigning the nodes that go into matching a value of 0.9 and the ones that do not 0.1. The problem seemed to be resolved.

Trained on a 1000 MNIST graphs model resulted on average with 55% of optimal weight possible. This was an expected result considering the limitation, but it at least

showed that model is gaining some knowledge compared to untrained model as well as a solution that randomly picks edges, where both resulted in 50-51% of optimal.

A model that barely beats random solution is not very usefull. Poor performance for now is mainly due to the very limited training, but before adding more data and prolonging training, we can experiments with the parameters of the network to see if they make any impact.

4.1.2 Model improvements and data augmentation

At this stage model seemed to at slowly improve and learn, but before trying to train it on the whole dataset there were still other techniques and parameters worth testing. Each of the techniques was tested separately.

It would make sense to start with the depth and width of the network. After trying to add up to 6 layers, it showed that more than 3 layers did not have any significant effect. Note: layers tell how many generations of neighbors is used for representation of a node. Increasing breadth did help, but impact got smaller as the breadth got wider. It would also impact running time so it was decided to have the model with 2 layers and 640 nodes at each layer (except input/output layers). Results were, however still rather low at 58%.

Next step was to adjust optimizer parameters: learning rate and weight decay. Too high learing rates resulted in too unstable training with information loss jumping too much, which indicated that model made too big adjustments after each training iteration. Too low learning rate did not give enough progress and made training proccess too slow. Value of 0.001 worked well as a middle ground.

Weight decay is another hyperparameter that can help with training the model. It can help to keep weight values relatively small and decrease chances of the model memorizing the answers, also called overfitting. However only negative effect were noticed even when training with larger datasets.

Adding skip connections to the architecture adds alternative way for the information to flow through the network. Skip connections help the model to reuse 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 harm in anyway and could be helpfull later.

Augmenting node features was another technique that was tested. The graphs were preproccessed Trying each additional feature separately to see it they have any benefits by themselves, then adding them one by one. Finally using all at once since they can mostly be calculated simultaneously and all have at least some positive effect:

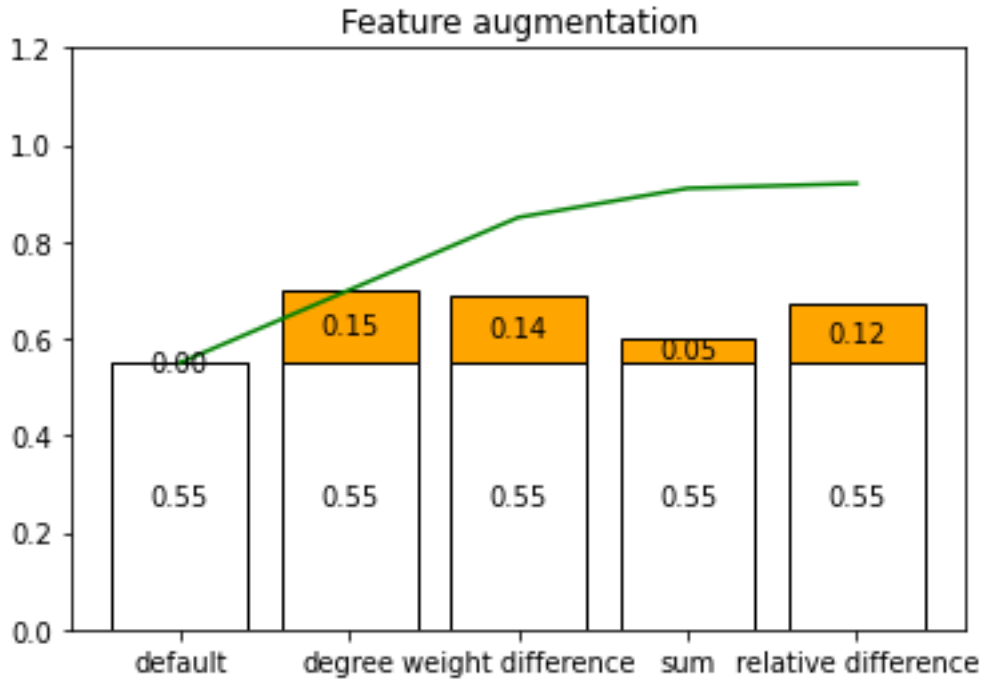

Figure 4.1: MNIST trained model. Average performance on 100 MNIST graphs

With fine tuning og the models hyperparameters and architecture, and augmenting node features having the most effect on the performance, results were moving closer to reasonable. At this point full training set can be used to see proper results. The results were relatively promising, but further testing showed that unfortunately line graph transformation on large and dense enough graphs turned out to expode in size and take to much time to proccess. Therefore another architecture was considered.

4.2 Edge prediction model

Instead of turning edges to nodes model can give predictions on the node pairs directly. With this approach there is less preprocessing needed, but the model now needs to have an

extra layer for edge prediction as described in Architecture section. After training both edge classification model and line graph model on the 55000 MNIST graphs following results were recorded: 101% for edge classification and 103% for line graph.

MNIST trained Model's performance on unseen MNIST graphs

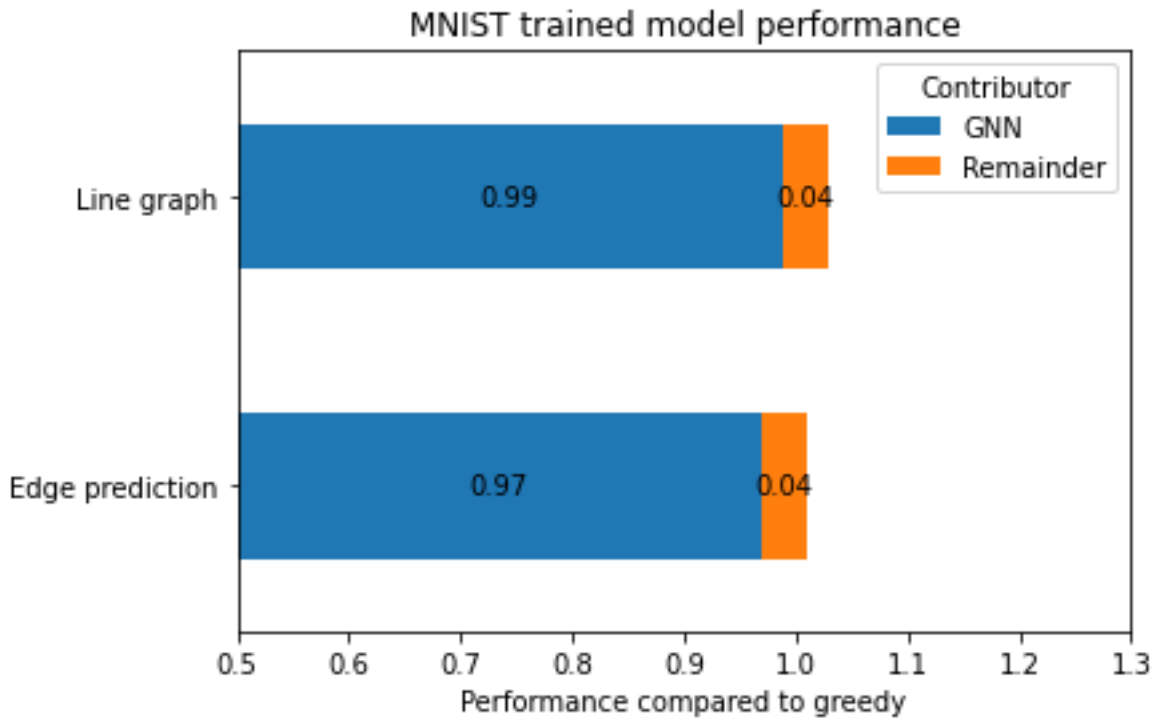

Figure 4.2: MNIST performance comparison

Edge classification showed a little bit worse results on average than the line graph. In theory it can be due to line graph containing more structural information in it compared to original. Regardless of the slightly worse result edge classification approach was still favourable due to time consumption of the line graph convertion. The results themselves were rather promising, both models manage to beat the greedy algorithm even when the greedy result is close to the optimal. However, there is still another probelm, time. Due to the small size of MNIST graphs finding optimal solution using Blossom algorithm takes less time than using this GNN model, because of the overehad computations needed for the neural network, but the model should be able to catch up timewise 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 ndoes and 100 000 edges and the model produces following results:

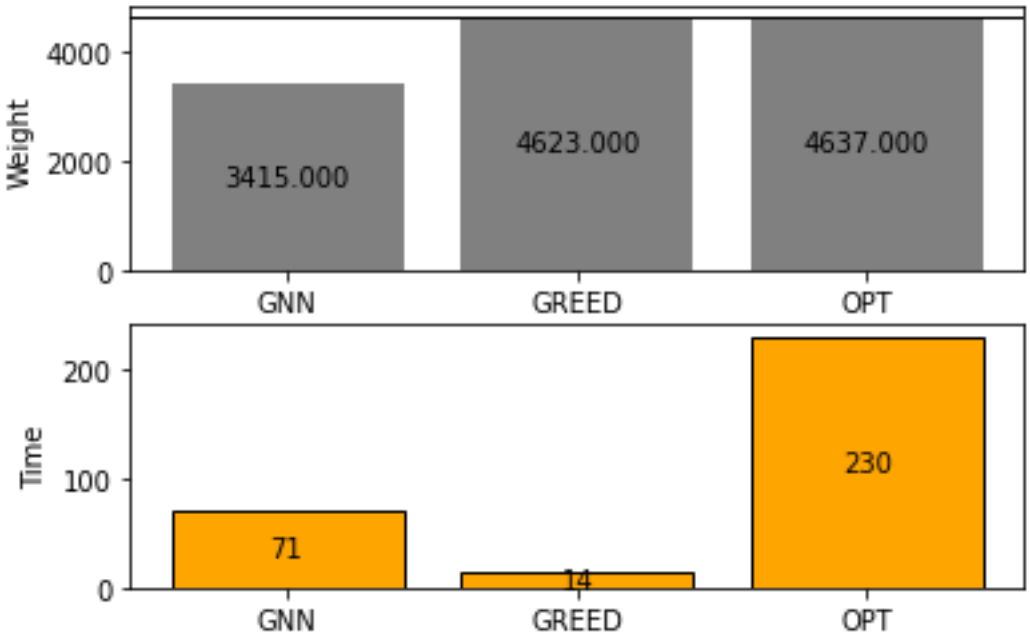

Figure 4.3: MNIST trained model performance on sage10 graph

As seen on the figure above GNN manages to beat the optimal solver in term 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 was to see if the GNN can be used on graphs larger than the ones it was trained on. It could be that model fails to learn what is needed for larger graphs. Another reason could be that MNIST graphs have similar structure 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 model gets during training the better. Therefore another dataset was tried 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 model struggled to learn the patterns. The flat information loss during training indicated that model had hard time to learn the more diverse collection of graphs. Adding additional layer to the classifier module of the model helped with stagnating learning curve. Adding even more layers did not give any noticeable difference like adding the first layer did. A slighly deeper than the previous model gave the following results on the new dataset:

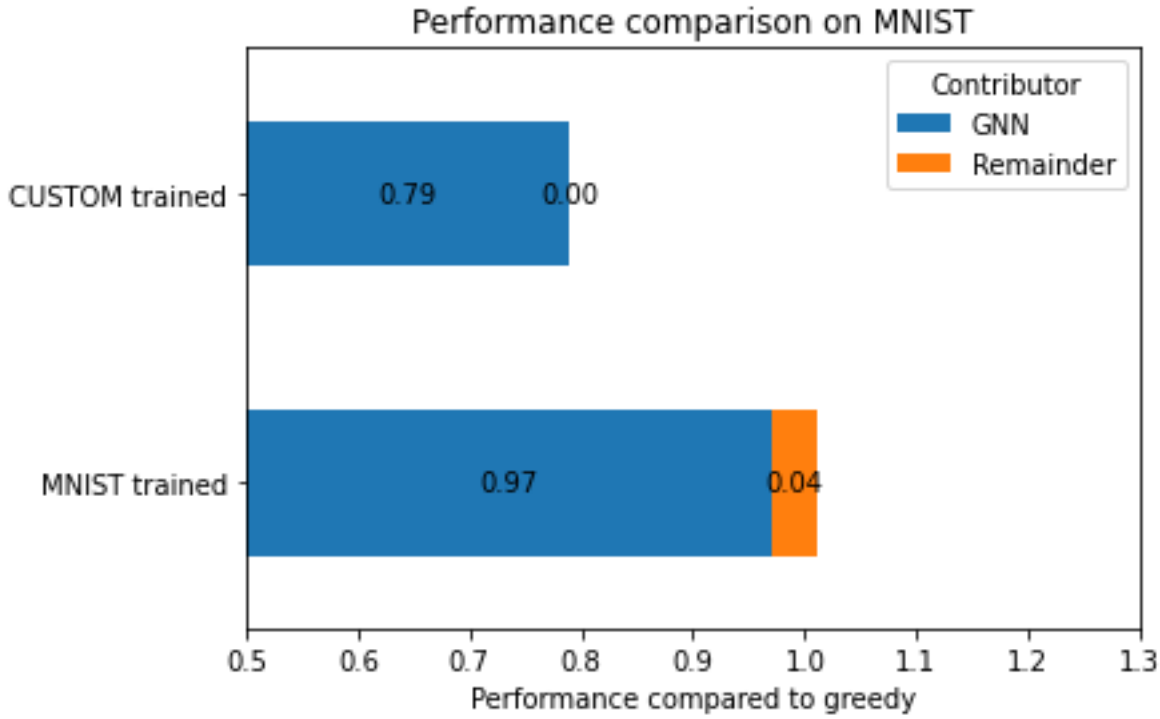

Figure 4.4: MNIST vs CUSTOM dataset training model performance on MNIST graphs

As partially expected model trained using custom dataset performed noticeably worse than the model trained on exclusively MNIST graphs. The custom dataset had both less graphs and larger graphs, than the ones in the MNIST dataset. This could be improved by adding some MNIST graphs to the custom dataset, but the main goal now is to see if the new dataset helps to get better performance on larger graphs.

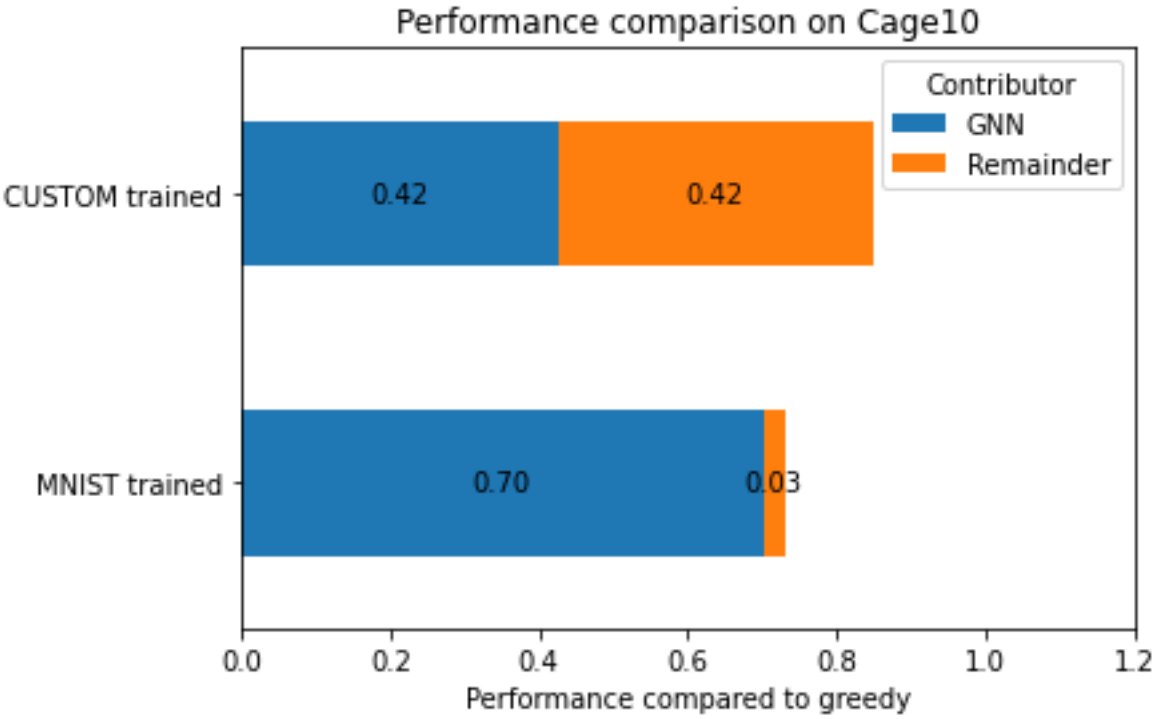

Figure 4.5: MNIST vs CUSTOM dataset training model performance on cage10

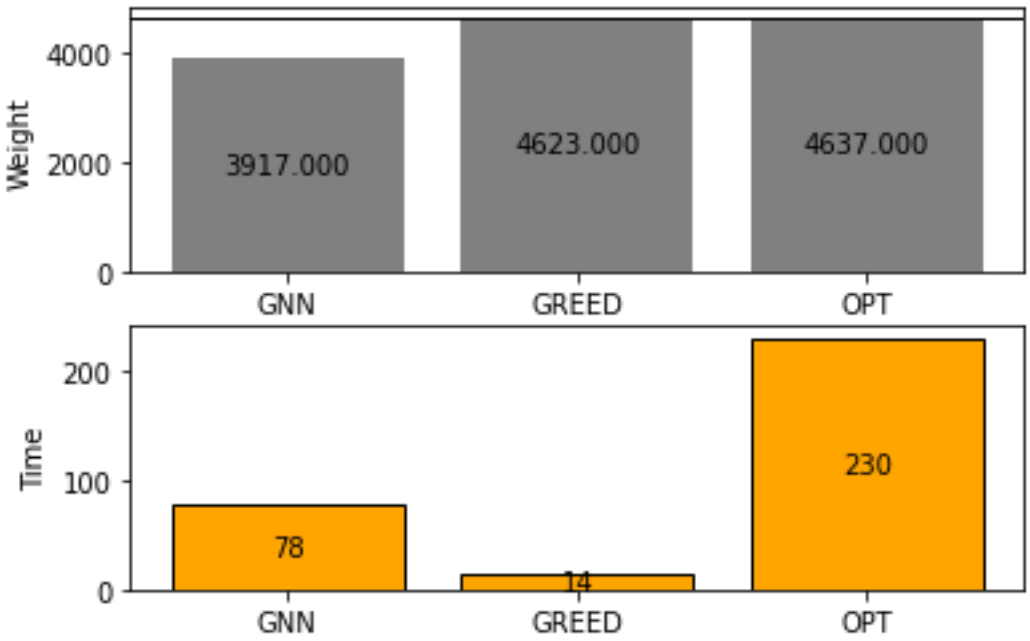

Figure 4.6: CUSTOM trained model performance on sage10 graph

The custom trained model does perform better in this case, however it is not neccesarily the models accuracy itself that is the reason of improvement. The model seems

to be rather indecisive on which nodes to match and half of the graph at the end is left unmatched opposed to the MNIST trained model matching almost the whole graph itself. This does however spark an idea that one might use the model to only match the most important nodes with highest probabilities and leave the rest to the greedy algorithm to achieve better results.

Results are still unsatisfying and there is a couple other ideas left to try: adjusting matching threshold for the model and applying reduction.

4.2.1 Reduction

Reduction rules are used in algorithms to cut down the size of the graphs by removing the easily recognizable parts of the graphs. Ideally the neural network should be able to pick up such, but it is still worth testing to see if there is any positive effect as well as see if model manages to grasp such rules by itself. The reduction rule itself is rather simple. If two connected nodes have an edge with a weight larger than the weight of the largest outgoing edge of each node there is no raeson to not pick it, since it is impossible to get a larger weight form these nodes.

Figure 4.7: Reduction example

A couple different graphs were used to test if the reduction helps and whether the model could find reducible nodes on its own. On some cases there were graphs that had 0 reducible edges, but in other cases such edges made around 5% of the total amount. The model however did not seem to pick any of those edges and adding reduction as a preprocessing step did improve the results by a small margin. Since the reduction is based

on the largest edges of the nodes it would make sense to add these node features to the model. New model was trained with 2 additional node features: 1st largest weight, 2nd largest weight for each node. Adding such features could have in theory helped the model to find reducible nodes, but after training the new model with the new node features the result did not improve.

4.2.2 Matching threshold

Adjusting the threshold for picking the edges was another thing to test as the previous results suggested. The model was initialy set to only consider edges that it predicted to have a larger than 50% chance of being part of the solution. This mathing treshold can be adjusted to for example only pick the edges with 90% probabilty, this might focus on the edges that have large impact on the total result, but would be otherwise ignored by a greedy algorithm. Additionaly it can be worth testing removal of edges with too low probabilty to see if model can find edges that impact the result negatively.

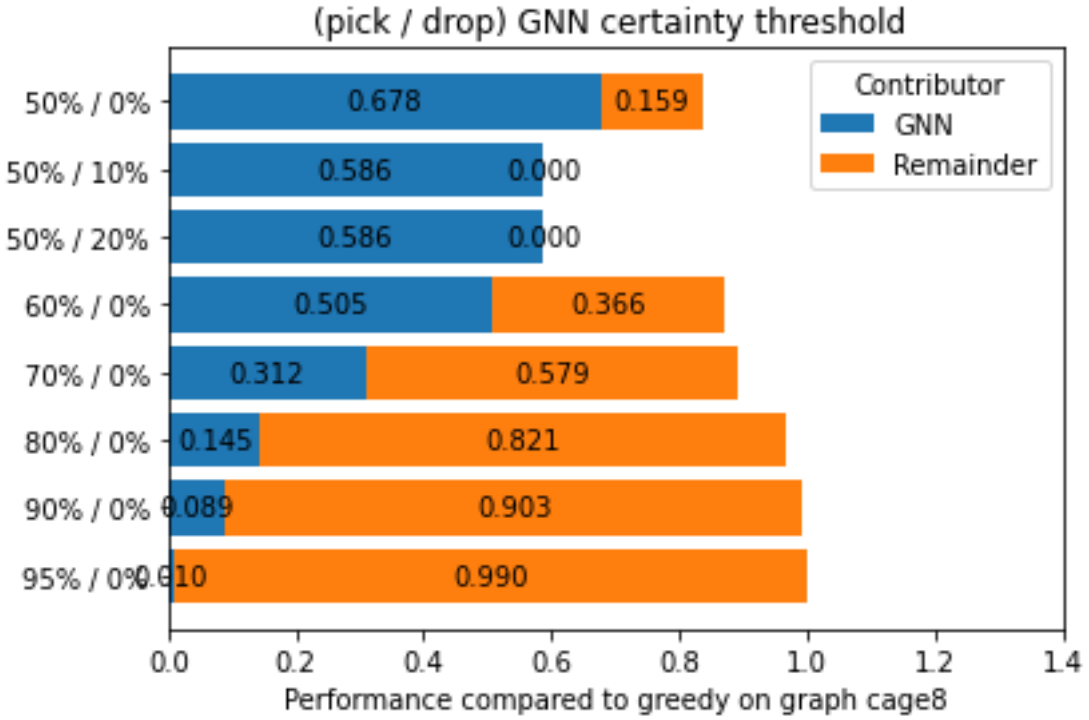

Figure 4.8: Model threshold test on cage8 graph

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

due to fewer edges being above the treshold. 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 results and no remainder to compensate. The reason for that is probably the weight class hyperparameter used during training which makes the model to care less for correctly finding nodes that should be ignored. The better approach could be to train model specifically to find high value edges.

4.3 Final results

With time and ideas for further improvements starting to end, 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. Epochss = 100
3. Network depth and width = 4 layers, 640 neurons each
4. Class weights = 0.1 and 0.9 for dropped and picked edges respectively
5. Weight decay = 0
6. Match threshold = 70%
7. 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.

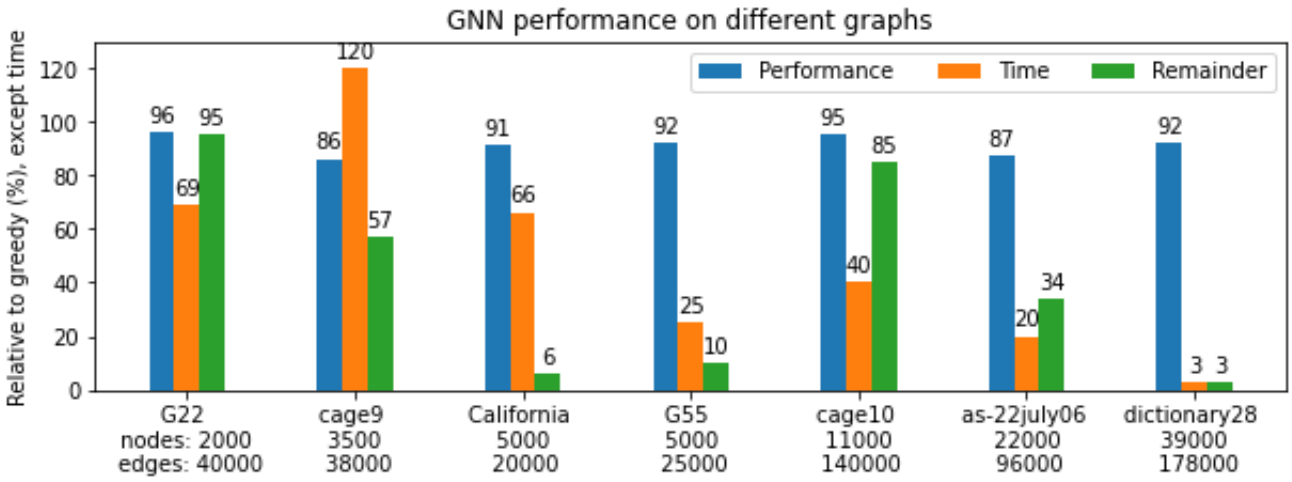

Figure 4.9: Final model performance

Lets summarize the end results based on the main criterias.

- Performance: The total weight the models output seems to be rather stable at around 90% of what greedy algorithm produces regardless of the size of the graph. It seems to be rather challenging for the model to beat the greedy appreach when in most cases in makes up around 90-95% 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. As expected the benefit of the using 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.
- Remaineder: 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 greedy algorithm

On average greedy algorithm seems to show really good results, but in theory it is not hard to make a graph that abuses the greedy approach and results in poor performance. Example of a graph that GNN model should be able to beat:

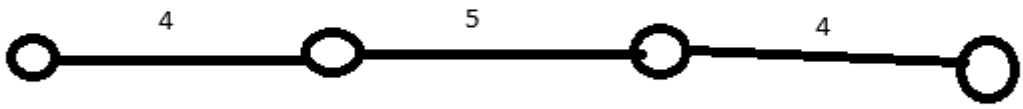

Figure 4.10: Good case for GNN

In this very simple case the greedy algorithm will result in total weight = 5. Yet the GNN does manage to get the optimal weight = 8. This ofcourse does not neccesarily prove anything, but shows that there might be use cases for the model. There are more realistic graphs that the model also manages to solve better. 1138bus graph from SuitSparse is one of the examples where GNN outperforms greedy by 15%. Additionaly, given that the runing times of greedy and GNN combined are still lower than running time of finding the optimal solution, one may even run both and choose the better result.

Chapter 5

Conclusion

Results show that GNNs are capable of solving MWM, but the approaches presented in this work showed worse performance overall comapared to a simple standard greedy algorithm. The margin between the results was not big enough to indicate that the approach was completely sensless. Compared to the greedy algorithm model showed some level of "understanding" of the task at hand and in some special cases even manages to beat the greedy algortihm. It is worth mentioning that theese cases were manualy chosen because of the way they abuse the naiveness of the greedy algorithm, but it does still indicate that such cases do exist and therefore there is value in using GNN instead of a greedy algorithm.

5.1 Future work

The fact that GNN in this work underperformed does not neccessary mean that the GNNs are in general unfit for MWM problem. There several potential improvements at hand. A deeper or wider model can be trained, meaning adding more layers as well as neurons to each layer to potentialy improve models ability to recognise complex patterns at the cost of longer training and prediction times. However for this particular architecture adding more layers showed little to no effect. Theres also a variety of different architectures that can be tested. A semi-supervised or an unsupervised approach is a good potential candidate where precomputing the optimal solution would not be needed. Instead the model can try to find the best solution by incentivising it to get as high weight sum as possible, resembeling a game in a way .

5.2 Final words

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

CO Combinatorial Optimization.

GCN Graph Convolution Network.

GNN Graph Neural Network.

MIS Maximal Indepndent Set.

MWIS Maximum Weighted Indepndent Set.

MWM Maximum Weighted Matching.

VCS Version Control System.

Bibliography

[1] 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`.

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

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

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

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

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

[7] 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`.

[8] 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`.

[9] 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.

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

[11] 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.

Appendix A

Code examples