CO 759 Project Report

Using Neural Networks for Variable Selection in the Branch and Bound Algorithm for the Traveling Salesman Problem

Marius Gutzeit, Philip Hodges and Zhiwei Shang

Instructor: Professor William Cook

University of Waterloo
Department of Combinatorics and Optimization

Abstract

In this report, we use deep learning to learn a branching rule in the branch and bound algorithm for solving traveling salesman problem (TSP) instances. We use a learning model that is independent of the size of the instance and learns to extract features from the TSP graph itself. In this way, a learned branching rule can be applied across different TSP instances. Our results indicate that properly learning an effective branching rule without more significant feature engineering outside of the TSP instance itself is more difficult than anticipated.

1 Introduction

Machine learning techniques have recently been applied to several areas of combinatorial optimization ([2], [3], [6], [7], [8]). As the study of machine learning matures, more areas of discrete optimization that are less well understood become apparent targets for machine learning to provide improvements. In this project, we explore how machine learning techniques may be able to improve the branch and bound algorithm, an essential component of solving mixed-integer programs (MIP). A survey of recent efforts in this area can be found in [1]. A significant aspect of the branch and bound algorithm is the decision, at each node, of what variable to branch on next, known as the branching rule. A considerable concern with branch and bound, being an exponential-time algorithm, is minimizing the size of the branch tree produced. Different branching rules will lead to potentially vastly different sizes of branch and bound trees, and so can make the difference in making the algorithm tractable or not.

Since the mathematical theory behind branching rules is poorly understood, machine learning is an approach to the problem. Past efforts have been focused on branch rules in general MIPs problems, and have made heavy use of manual feature extraction. See, for example, [2], who use 72 different features from 11 categories that seem to be reasonable predictors of good branch rules. This approach has been successful, but we consider approaching the problem more in the style of deep learning. That is, instead of manually choosing a set of features, we simply provide basic general data to a neural network, which will then learn to determine essential features automatically. Furthermore, we restrict ourselves to a specific class of MIPs, the traveling salesman problem (TSP). This allows us to capture features of the graph defining the TSP instance with a graph embedding method, which was used in [3] for learning greedy algorithms in combinatorial optimization problems. This graph embedding methodology gives a constant size feature vector for the graph, so the neural network given by the graph embedding can be trained on TSP instances of varying sizes. This is a desirable feature if we are aiming to learn a branching rule that works well across different TSP instances.

In Section 2, we describe the branch and bound framework we used for the learning context and the graph embedding model we trained to learn a branching rule. In Sections 3 and 4, we describe the training process and evaluate how well the trained model performed.

2 Methodology

In this Section, we elaborate on the used graph embedding neural network which was modified form an existing deep learning architecture proposed by Dai *et. al*, [3]. Furthermore, we define the branching rule and how we generate the data we use.

2.1 Strong Branching in Branch and Bound

In order to have our graph embedding framework learn a branching rule, we require some way to determine what a reasonable branching rule is. Before we discuss this, we give pseudocode for a branch and bound in Algorithm

1. In this pseudocode, the branch nodes N_i are defined by the set of variable constraints produced from previous branches, and the priority queue used is ordered by the value of the LP solution stored at that node. Having the algorithm accept an initial solution as a parameter gives us the option of starting out with a known optimal solution; this is desirable because, having fixed both the node selection procedure and the tree pruning bound, the only factor determining branch tree size is the branching rule.

Algorithm 1 Branch and Bound

```
1: procedure TSPBranchAndBound(G, initial solution, initial solution value)
2:     best solution value ← initial solution value
3:     best solution ← initial solution
4:     PriQueue.AddNode(N_0, None, None)
5:     while PriQueue not empty do
6:         (N, LP solution, Obj value) ← GetNode(PriQueue)
7:         if LP solution is a tour AND Obj value < best solution value then
8:             best solution value ← Obj value
9:             best solution ← LP solution
10:        (N_1, N_2) ← BranchRule(N, G)
11:        for i = 1, 2 do
12:            (LP solution, Obj value) ← LowerBound(N_i, G)
13:            if Obj value < best solution value then
14:                PriQueue.AddNode(N_i, LP solution, Obj Value)
```

In order to fully specify the algorithm, we also require a branch rule and a lower bounding method. The typical lower bounding method used for the TSP involves solving the degree LP relaxation

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} w_e x_e \\ \text{subject to} \quad & x(\delta(v)) = 2, \quad \forall v \in V \\ & 0 \le x_e \le 1 \quad \forall e \in E, \end{aligned}$$

and add subtour cutting planes and other types of cutting planes (resulting in a "branch and cut" algorithm). In the interest of producing larger branch trees so that data collection is easier, we use a simpler lower bounding technique, which is to solve the LP relaxation and add only enough subtour cutting planes to get a connected solution.

The branch tree is now fully determined by the branch rule. Thus a good branching rule is one that branches in the order that produces the smallest branch tree. However, this order is computationally impossible to determine in practice, and so instead, a branch rule called Strong Branching (SB) is typically used (e.g. in [2]). For each candidate variable for branching, we compute a SB score based on how good that branch would be. This score is based on the changes in the lower bound of the child nodes in the branch and bound process; if the lower bound increases a lot, this is heuristically likely to lead to a smaller branch tree. With this in mind, at each node N of the branch tree and each candidate edge e for branching, we use the SB score

$$SB_{N,e} = \max(LB_0 - LB, 0.1) \cdot \max(LB_1 - LB, 0.1),$$

where LB is the lower bound at the node N, and LB_0, LB_1 are the lower bounds for the child nodes with $x_e = 0$ and $x_e = 1$ respectively.

As shown in our results, this branch rule produces much smaller branch trees than other more straightforward rules, but the computations required at each node are significant. We wish to train our graph embedding network to identify edges in the graph that produce high SB scores, so we can produce similarly sized branch trees with far less computation.

2.2 Graph Embedding Neural Network

In [3], Dai *et. al* leverages a novel deep learning architecture over graphs, in particular, the *structure2vec* of [4], to parameterize the evaluation function, $\hat{Q}(h(S), v; \Theta)$, with parameters Θ in the following way.

The graph embedding network, *structure2vec* will compute a $p-$dimensional feature embedding μ_v recursively for each node $v \in V$, given the current partial solution S using the following formula:

$$\mu_v^{(t+1)} \longleftarrow \text{relu}\big(\theta_1 x_v + \theta_2 \Sigma_{u \in \mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \Sigma_{u \in \mathcal{N}(v)} \text{relu}(\theta_4 w(v,u))\big), \tag{1}$$

where $\mathcal{N}(v)$ is the set of neighbours of node v in graph G, $\theta_1 \in \mathbb{R}^(p), \theta_2, \theta_3 \in \mathbb{R}^{p\times p}$ and $\theta_4 \in \mathbb{R}^{p\times p}$ are the models parameters, x_v is an indicator variable of node v in the partial solution S and $w(u,v)$ is the weight of edge $e(u,v)$, and relu is the rectified linear unit ($\text{relu}(z) = z$ if $z > 0$ and 0 otherwise) applied elementwise to its input.
The evaluation function is defined as:

$$\hat{Q}(h(S), v; \Theta) = \theta_5 relu([\theta_6 \Sigma_{u\in V} \mu_u^{(T)}, \theta_7 \mu_v^{(T)}]) \tag{2}$$

where $\theta_5 \in \mathbb{R}^{2p}, \theta_6, \theta_7 \in \mathbb{R}^{p\times p}$ and $[\cdot,\cdot]$ is the concatenation operator.
Inspired by the above idea, we propose our own deep learning graph embedding model for branching variable selection in branch and bound algorithm of TSP. Correspondingly, our iteration process and evaluation function are defined as follows:

$$\mu_v^{T+1} = \text{relu}\left(\theta_1 \frac{\sum_{e\in\delta(v)} \text{relu}(\theta_2 \bar{x}_e)}{n} + \theta_3 \frac{\sum_{u\in\mathcal{N}(v)} \mu_u^T}{n} + \theta_4 \frac{\sum_{u\in\mathcal{N}(v)} \text{relu}(\theta_5 w(u,v))}{n}\right) \tag{3}$$

where $\theta_1, \theta_3, \theta_4, \in \mathbb{R}^{p\times p}, \theta_2, \theta_5 \in \mathbb{R}^p$ for some finite p, and $\bar{x_e}$ is the value of the variable corresponding to e in the current branch and bound LP solution which could be fractional; $\delta(v)$ is the set of neighbours of node v in the current fractional solution to the TSP, more specifically, if the edge $e(u,v)$ is non-zero in the fractional solution, $v \in \delta(u)$; $\mathcal{N}(v)$ is the neighbours of v in the graph; n is the total number of node in the current graph

$$\hat{Q}(N, e=(u,v), \Theta) = \theta_6 \text{relu}\left(\frac{\left[\theta_7 \Sigma_{w\in V} \mu_w^T, \theta_8 \frac{(\mu_u^T + \mu_v^T)}{2}\right]}{n}\right) \tag{4}$$

where $\theta_7, \theta_8 \in \mathbb{R}^{p\times p}$, and $\theta_6 \in \mathbb{R}^{2p}$;$[\cdot,\cdot]$ and n are of the same meaning as in 3.

The main modifications between our model and the model in [3] are the divisor n and the first item in μ iteration. The reason for adding divisor n is simple since the summation in the iteration process tends to grow infinitely if the total number of nodes is very large; empirically, the branching variable should be chosen from the fractional variables in the current branch and bound solution, so we encode this information into the graph embedding structure by modifying the first item from [3] to $\left(\Sigma_{e\in\delta(v)} \text{relu}(\theta_2 \bar{x_e})\right)/n$.

Another big difference between our model and the model of [3] is the method of training. We require a ground truth label for every input graph G in order to train the *structure2vec* architecture which is very similar to [4]; the output of the embedding is linked with a softmax-layer, so that the parameters can be trained end-to-end by minimizing the cross-entropy loss.

Our new model is also n (the total number of nodes) independent. So we can train our model using data generated from various sizes of TSP instances and use it to make predictions for TSP instances of different sizes.

2.3 Data Generation

To collect data for use in the graph embedding, we first generated a set of random Euclidean TSP instances. That is, nodes were chosen by random integer coordinates in a 1000×1000 space, and edge weights are computed as the Euclidean distance between the points rounded to the nearest integer. We produced 10 graphs of each size for 50 nodes up to 100 nodes, in 5 node intervals. If the branch and bound algorithm using SB as a branching rule took more than 15 minutes to complete, we discarded the instance to maintain low computational requirements. This procedure produced 95 graphs, of which 68 were allocated to training and 27 allocated to testing.

For each generated instance, we ran our branch and bound algorithm with SB. At each node, the LP solution at that node and the graph are saved, as well as a label related to the strong branching score for each candidate variable for branching. Thus each variable represents a data point that the network can learn from.

The labels are generated in a similar fashion as in [2]. For each node, we compute the largest SB score $SB_{N,e}$. We assign all variables $x_{e'}$ with a score $SB_{N,e'} \geq 0.8 \cdot SB_{N,e}$ a label of 1, and all other variables a label of zero. This is to avoid the neural network learning to differentiate between two poor branching variables; we really only require that it is able to distinguish a "good" variable from a "bad" one. We found that this method produced quite a large number of 0 labels, as many variables produced no change or almost no change in the lower bound. We thus removed a random selection of 0 labels at each step to bring the proportion of 1 labels to 0 labels closer to 1:4.

3 Training

3.1 Environment

All computations were run on a Linux system with an Intel Core i5-6600K 4.6 GHz CPU and an NVIDIA GTX 1070 GPU. Memory and storage requirements were small enough to not significantly impact computational time. Implementation of the algorithms and network was done in Python, using PyTorch for learning functionality. The code can be found in [5].

3.2 Training Algorithm

Our model is implemented as a binary classifier and the training data are a set of (*data, label*) pairs. The neural network is trained end-to-end by minimizing the cross-entropy loss.

Algorithm 2 Training Algorithm

```
Optimizer = ADAM(graph_embedding_nn, lr)
for for epoch e = 1 to L do do
    while (data, label) = load_batch_from_training_set(bt_size) && len(data) ≥ bt_size do
        output = graph_embedding_nn (data)
        loss = CrossEntropyLoss(output, label)
        do loss backpropagatation
        Use Optimizer to update Θ
return graph_embedding_nn
```

3.3 Training Loss

We trained our model for 600 epochs with learning rate = 0.001, embedding iteration number $T = 3$, batch size = 6, and feature vector length = 25. These parameters were selected largely based on the computational requirements of training. The model was trained on the 68 TSP training instances whose sizes vary from 50 to 100 nodes. This resulted in a total of approximately 10000 data points.

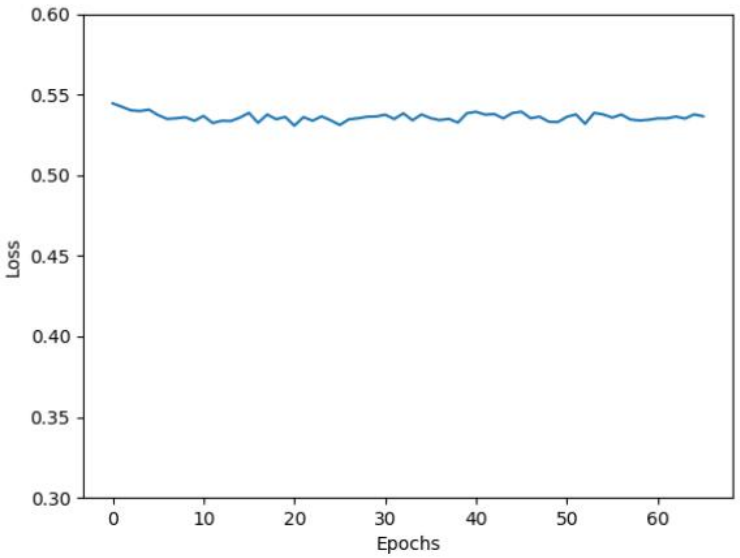

Figure 1: Training Loss for the first 60 epochs

The training loss graph for the first 60 epochs is shown in Figure 1. We can observe that the loss stops decreasing after about 10 epochs. There are many possible reasons:

- The feature-length p is too small to describe the complexity of the graphs.

Dataset	Precision	Recall	Accuracy
Training Data	0.251	0.251	0.648
Testing Data	0.247	0.247	0.644

Table 1: Precision, Recall and Accuracy

- The initial setting of hyper-parameters is sub-optimal which results in the neural network gets stuck in a local optimum.
- The designed neural network cannot learn the strong branching rule at all. A more customized graph embedding neural network might be needed to approximate the strong branching rule properly.

However, it is difficult to determine what the real reasons are, and any combination of these changes and others might be needed to improve learning.

4 Evaluation

The evaluation environment is the same as in the training process.

There are a few differences between how we use our model in training and testing. In training, we load a random batch of data (*(edge, label)* pairs) from the training dataset, and then output two scores indicating the probability of each edge being label 0 or label 1. In testing, instead of loading a random batch of data, we load all candidates for branching in an inner node of a branch and bound branch tree and then return a list of scores for being label 1 one for each edge. Finally, we impose that the first K candidates in the predicted list are label 1 where K is the number of candidates that are indeed labeled as 1 by the strong branching rule at the same branching node.

4.1 Precision, Recall and Accuracy

The precision, recall and accuracy of the model tested on both training dataset and testing dataset are listed in Table 1.

Since we force the number of label 1 in the predicted results to be the same as the number of label 1 in the strong branching dataset, the precision and recall will always be the same as each other. The numbers for training data and testing data are very close to each other indicating there is no overfitting. However, the precision and recall are only around 25%, it indicates that only about 25% of the 1-labeled candidates are found by the trained model, which is very low. The accuracy is better than 0.5, however, this is the result of how we label the predicted data since we are always keeping the number of predicted labels of 1 the same as in the training data. Thus, the accuracy being better than 0.5 indicates nothing in addition of what the precision shows.

4.2 Applying the Learned Branch Rule

We tested the learned branching rule by employing it in the branch and bound algorithm described in section 2. In particular, the embedding network produces the ratio of the probability that a variable is classified as a 1-label to the probability it is classified as a 0-label for each variable. The variable with the maximum value is then taken as the branch variable.

We have two measures of performance quality of a branching rule in the algorithm:

- The number of nodes in the branch and bound tree; this is a reasonable measure of how well the network learned to approximate the strong branching rule. A smaller tree shows that the learned rule was better able to select variables to branch on that would lead to tree pruning. While we would not expect a learned branching rule to perform as well as strong branching in this regard, we can deduce how well the learning process worked from how closely the number of nodes in the branch tree for the learned rule comes to that of the strong branching rule.
- The time taken for the algorithm to run to completion; evidently, a faster algorithm is better, and while a smaller branch tree can generally lead to a faster running time, the strong branching rule is quite slow, and so can take longer to run than other rules on some instances. If a learned rule can reasonably approximate the size of the branch tree of strong branching, then the enormously decreased time spent on the branch decision would result in a much faster algorithm.

We ran the branch and bound algorithm with five different branching rules on the 27 TSP instances generated for testing. The branching rules include both strong branching and the learned branching rule, as well as:

- Random branching: of the variables assigned a fractional value in the LP solution, choose one at random
- Most fractional branching: select the variable that has a value in the LP solution closest to 0.5
- Largest objective branching: choose the variable with largest corresponding edge weight in the graph (coefficient in the objective function)

The average number of branch tree nodes and average runtime of graphs of the same size are plotted against the size of the graph in Figures 2 and 3. The branch tree size is plotted on a logarithmic scale.

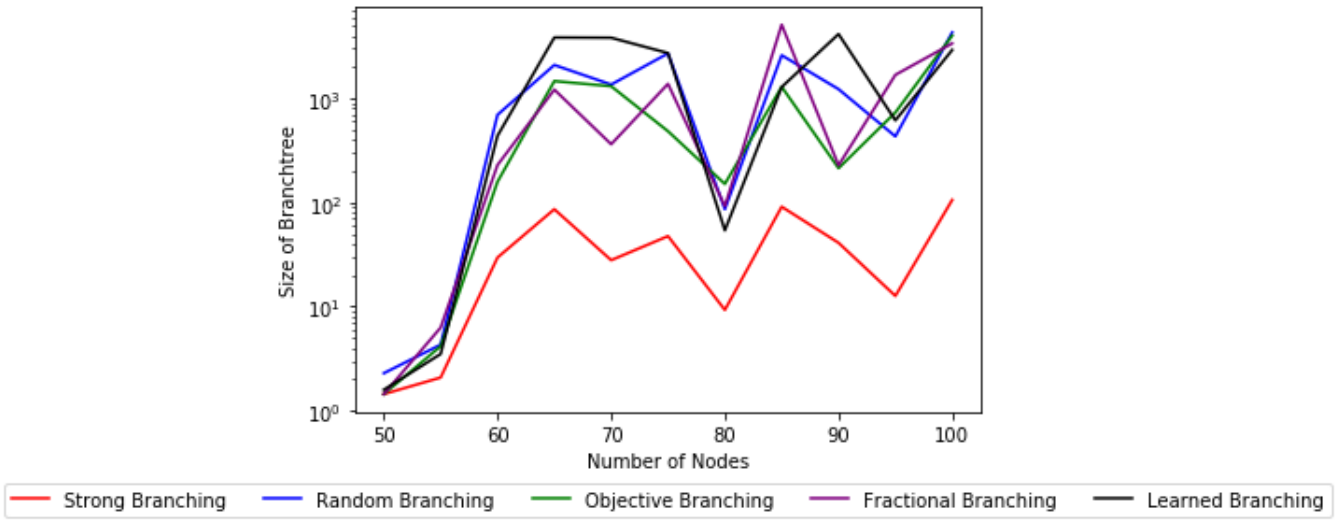

Figure 2: Branch tree sizes of branch and bound as a function of graph size on test graph instances

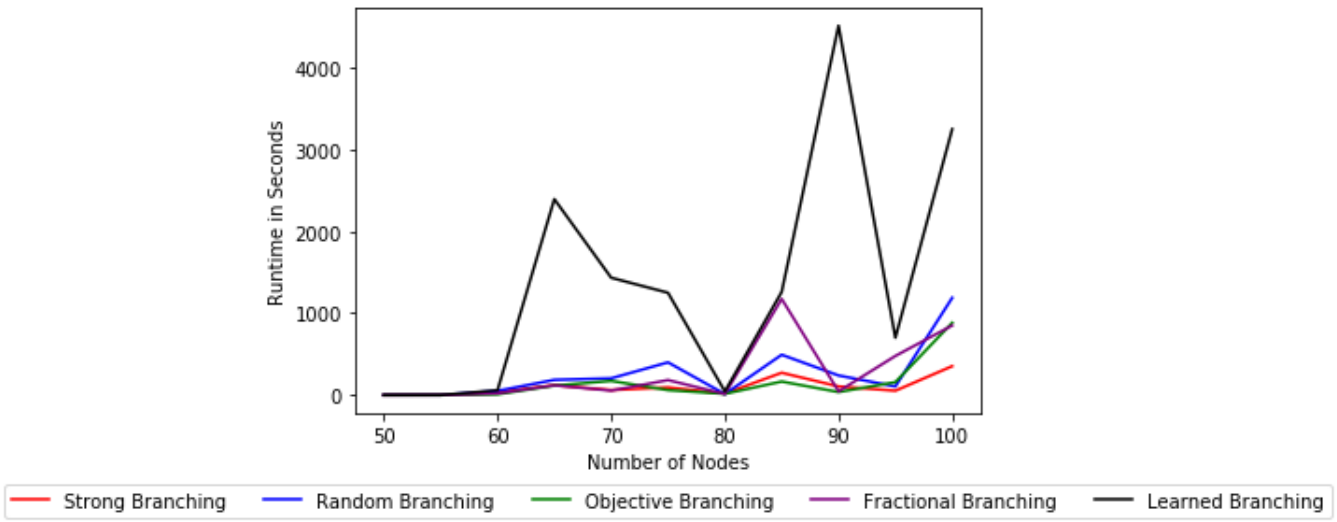

Figure 3: Runtimes of branch and bound as a function of graph size on test graph instances

As can be seen from Figure 2, the learned branching rule was unable to attain appreciably smaller branch trees than a random branching rule, which performed on average the same as the remaining branching rules excepting strong branching. Regarding runtime, this means that, as seen in Figure 3, the learned branch rule was significantly slower than the other branch rules when the branch tree size becomes significant, as strong branching tree sizes make it the faster algorithm while the remaining rules require much less time to select a branch variable.

We also tested the learned branching rule on a few instances from the TSPLIB, with similar results. These results are plotted in Figures 4 and 5.

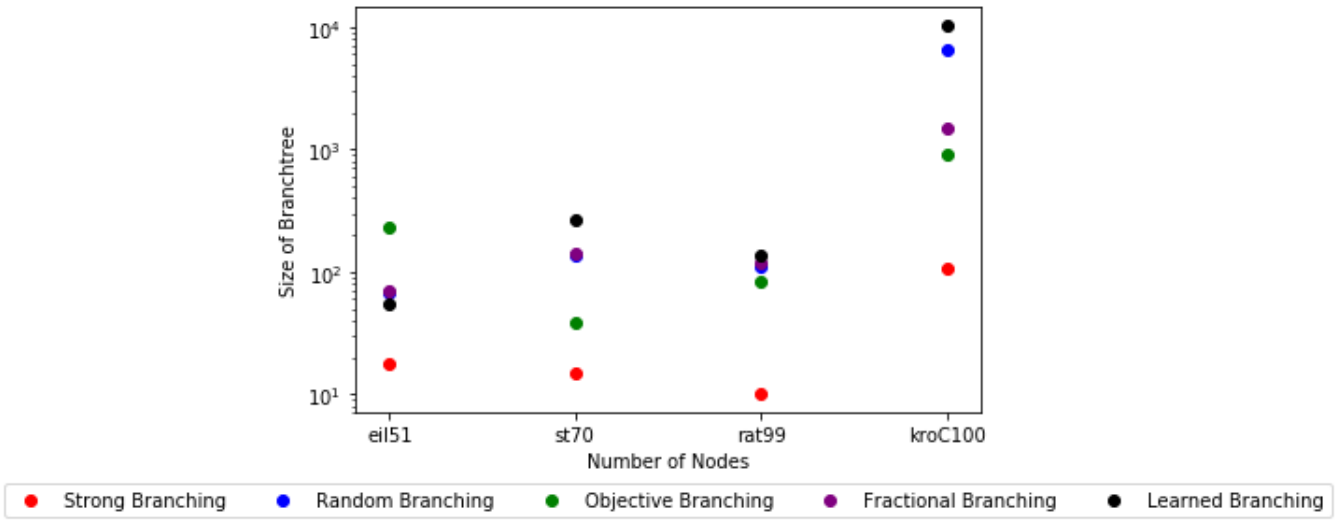

Figure 4: Nodes of branch and bound on selected TSPLIB instances

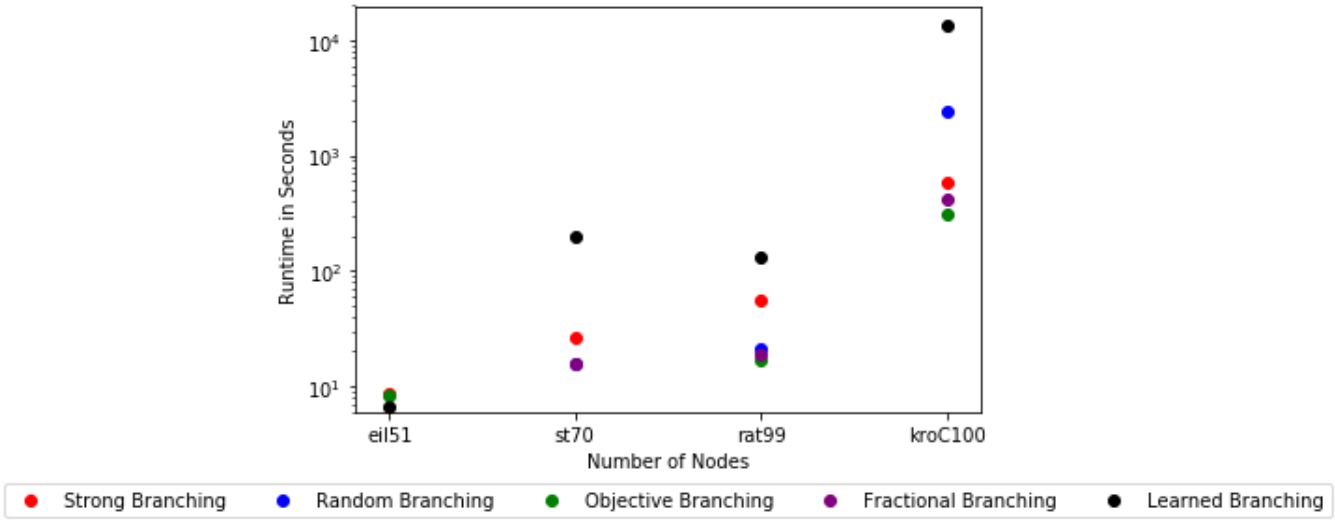

Figure 5: Runtimes of branch and bound on selected TSPLIB instances

These results quite conclusively show that the trained network was unable to identify features that help with meaningfully predicting good branching variables. Ideally, the branch tree sizes of the learned rule would have been smaller than those of the three simple branching rules, and not too much larger than the branch trees produced by strong branching. In this way, the learned algorithm would be faster than the basic branching rules by virtue of a smaller branching tree, and also faster than strong branching by virtue of faster processing at each node of the tree.

It is a possibility that information about the graph given by the embedding formulation that we used is simply not sufficient to train a good branching rule. Future attempts at this kind of learning methodology should perhaps include more features, such as the dual LP solution at each branch tree node, as well as perhaps features related to the constraints used for lower bounding at each node.

5 Conclusions

Our results show that our proposed learning model was unable to effectively learn to imitate a strong branching rule. It is unclear whether the issue lies in the effectiveness of the specific graph embedding framework we applied, the parameters used for training, or more fundamentally in the impossibility of determining a branching rule solely from a graph. Applying the learned branching rule in a branch and bound algorithm to a test set of random Euclidean TSP instances confirms that the learned branching rule was unable to identify features of a good branching variable. Thus the problem of learning a branching rule on TSP instances with minimal feature engineering is a more difficult task than it initially appears.

6 Future Work

In subsequent work, an alternative to the graph embedding neural net used here should be considered. The results of the loss function in particular indicate a need for a more customized neural net to feed deep learning models with an appropriate amount of information about the graph. Since our goal was to use the least amount of features necessary, it should be evaluated what other features, like the dual LP solution, will improve performance when added to the model. Furthermore, future studies should carry out more extensive testing with a variation of hyper-parameters for the neural net as well as training on a larger dataset. To this end, different data generation methods should be explored.

References

[1] Andrea Lodi, Giulia Zarpellon. *On learning and branching: a survey.* 2017.

[2] Elias B. Khalil, Pierre Le Bodic, Le Song, George Nemhauser, Bistra Dilkina. *Learning to Branch in Mixed Integer Programming.* AAAI 2016.

[3] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, Le Song. *Learning Combinatorial Optimization Algorithms over Graphs.* NIPS 2017.

[4] Hanjun Dai, Bo Dai, and Le Song. *Discriminative embeddings of latent variable models for structured data.* ICML 2016.

[5] Philip Hodges. branchlearning, https://github.com/TrivialError/branchlearning.

[6] Wouter Kool, Max Welling. *Attention solves your TSP* March 2018.

[7] Mohammadreza Nazari, Aafshin Oroojlooy, Lawrence V. Snyder, Martin Takac. *Deep reinforcement learning for solving the vehicle routing problem.* February 2018.

[8] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, Samy Bengio. *Neural combinatorial optimization with reinforcement learning.* 2017.