

---

# ML4CO: Is GCNN All You Need? Graph Convolutional Neural Networks Produce Strong Baselines For Combinatorial Optimization Problems, If Tuned and Trained Properly, on Appropriate Data

---

**Amin Banitalebi-Dehkordi**  
Huawei Technologies Canada Co., Ltd.  
amin.banitalebi@huawei.com

**Yong Zhang**  
Huawei Technologies Canada Co., Ltd.  
yong.zhang3@huawei.com

## Abstract

The 2021 NeurIPS Machine Learning for Combinatorial Optimization (ML4CO) competition was designed with the goal of improving state-of-the-art combinatorial optimization solvers by replacing key heuristic components with machine learning models. The competition’s main scientific question was the following: is machine learning a viable option for improving traditional combinatorial optimization solvers on specific problem distributions, when historical data is available? This was motivated by the fact that in many practical scenarios, the data changes only slightly between the repetitions of a combinatorial optimization problem, and this is an area where machine learning models are particularly powerful at.

This paper summarizes the solution and lessons learned by the Huawei EI-OROAS team in the dual task of the competition. The submission of our team achieved the second place in the final ranking, with a very close distance to the first spot. In addition, our solution was ranked first consistently for several weekly leaderboard updates before the final evaluation. We provide insights gained from a large number of experiments, and argue that a simple Graph Convolutional Neural Network (GCNNs) can achieve state-of-the-art results if trained and tuned properly.

## 1 Introduction

Combinatorial optimization has a wide range of applications in our day-to-day lives. Examples include: retail, manufacturing, data center management, energy systems, airline scheduling, auction design, political districting, kidney exchange, scientific discovery, ridesharing, cancer therapeutics, conservation planning, disaster response, and or even in college admission [6; 19]. Due to its importance, it has been under extensive research and explorations especially in computer science and mathematics. This has resulted in hundreds of research papers and many successful commercial and open-source solvers available to use [8; 11; 16; 1; 12; 7].

While most combinatorial optimization solvers are presented as general-purpose, one-size-fits-all algorithms, the NeurIPS 2021 ML4CO competition focuses on the design of application-specific algorithms from historical data [10]. This general problem captures a highly practical scenario relevant to many application areas, where a practitioner repeatedly solves problem instances from a specific distribution, with redundant patterns and characteristics. For example, managing a large-scale energy distribution network requires solving very similar CO problems on a daily basis, with a fixed power grid structure while only the demand changes over time. This change of demand is hard to capture by hand-engineered expert rules, and ML-enhanced approaches offer a possible solution to

detect typical patterns in the demand history. Other examples include crew scheduling problems that have to be solved daily or weekly with minor variations, or vehicle routing where the traffic conditions change over time, but the overall transportation network does not.

The ML4CO competition [10] features three challenges for ML, each corresponding to a specific control task arising in the open-source solver SCIP [12], and exposed through a unified OpenAI-gym API based on the Python library Ecole [9]. For each challenge, participants were evaluated on three problem benchmarks originating from diverse application areas, each represented as a collection of mixed-integer linear program (MILP) instances.

This paper summarizes the solution and lessons learned by the EI-OROAS team in the dual task of the competition. The submission of our team achieved the second place in the final ranking, with a very close distance to the first spot. In addition, our solution was ranked first consistently for several weekly leaderboard updates before the final evaluation. We provide insights gained from a large number of experiments, and argue that a simple Graph Convolutional Neural Network (GCNNs) can achieve state-of-the-art results if trained and tuned properly, on the right data. In addition, we provide a series of remarks and guidelines which we believe will be useful for practitioners to pay attention to, in order to get to the full potential of their NN models used for combinatorial optimization.

## 2 Background

Most combinatorial optimization problems in practice can be written as mixed-integer linear programs (MILP). Branch-and-bound (B&B) algorithm [20] is an exact method for solving MILPs. When interrupted before termination (or completion), B&B can provide intermediate solutions along with optimality bounds (even without actually knowing the optimal values) [20; 13]. This algorithm builds a solution tree and recursively selects variables for branching (LP relaxation). This process was traditionally done with highly tuned hard-coded heuristics, however, over the past several years several machine learning (ML) approaches are also emerging [13; 14; 22; 26; 17]. The idea is that for repetitive type of problems such as production planning, one may be able to use an ML model for processes such as variable selection (a.k.a. branching), in order to minimize the total solving time of MILP instances.

A mixed-integer linear program (MILP) can be expressed as follows:

$$\begin{aligned} & \arg \min_{\mathbf{x}} \mathbf{c}^\top \mathbf{x} \\ \text{s.t. } & \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p}, \end{aligned} \quad (1)$$

where  $\mathbf{c} \in \mathbb{R}^n$  is the objective coefficient vector,  $\mathbf{A} \in \mathbb{R}^{m \times n}$  is the constraint coefficient matrix,  $\mathbf{b} \in \mathbb{R}^m$ ,  $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ , and  $p \leq n$  is the number of integer variables. Moreover,  $m$  is the number of rows and  $n$  refers to the number of columns.

Relaxing the integrality constraint turns the problem into a linear program (LP) which can be solved (e.g. via the simplex algorithm) to obtain a lower bound to (1). Solution to the LP relaxation will be also a solution to (1), assuming it still respects the original integrality constraint. Otherwise, two sub-problems can be created on the two sides (floor and ceil) of an integer variable. This binary decomposition is repeatedly done via the branch-and-bound algorithms.

Fractional variable selection in branch-and-bound can be done in different ways. These methods will have a different impact on the size of the B&B tree, and therefore in the speed and performance of a solver [3]. Example methods include:

- strong branching (SB) [5]: results in smallest search tree but may be slow on iterations, as it will have to solve two LPs for each branching candidate.
- pseudo-cost: estimates the branching effect based on variable history [25]
- hybrid: to perform SB at the beginning and slowly switch to pseudo-cost or other simpler heuristics [14; 2]

Machine learning based solutions treat the variable selection and branching processes as a markov decision process, and try to employ learning algorithms as a replacement for conventional branching rules. Next we will briefly mention some of these ML-based strategies.

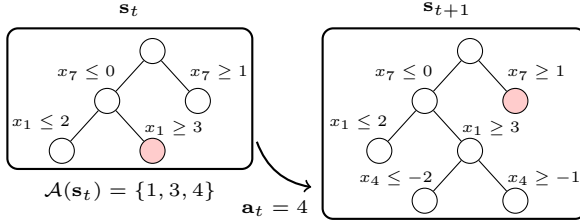


Figure 1: A demonstration of the B&B variable selection. Left: a node is selected by the solver to be branched in the next iteration. Right: a new state formed. Visualization from [13].

### 3 Related Work

Earlier works include methods that try to learn to mimic the strong branching rule. [4] defines a regression problem (predicting the SB scores) while [15; 18] approach it as a ranking problem (predicting the order of the candidates). [13] models the branching as a classification problem where labels are branching decisions from the SB rule. This GCNN-based approach for the first time reported results better than a solver (SCIP) that uses presolving, primal heuristics, and cuts. Later, [23] built on the GCNN method by incorporating an ADMM-based expert to scale-up the full strong branching to large instances. Other works include [26] where authors show that using the entire B&B tree can further boost the imitation learning, or [14] where the imitation learning was made faster by switching to a small MLP after the root node of the tree.

Our solution employs the GCNN approach of [13]. In particular, we show that the inherent incorporation of graph convolutional neural networks for this problem is a suitable choice and with proper tuning and training on suitable training data, it can still achieve state-of-the-art results.

### 4 The ML4CO Competition

**Challenges:** There are three distinct challenges, each corresponding to a specific control task arising in traditional solvers: primal, dual, and configuration. This article discusses the details of the submission by the Huawei EI-OROAS team on the dual task, that is about selecting branching variables, in order to minimize the dual integral over time.

**The dual task:** The dual task deals with obtaining tight optimality guarantees (dual bounds) via branching [10]. Making good branching decisions is regarded as a critical component of modern branch-and-bound solvers, yet has received little theoretical understanding to this day [21]. In this task, the environment will run a full-fledged branch-and-cut algorithm with SCIP, and the participants will only control the solver’s branching decisions. The metric of interest is the dual integral, which considers the speed at which the dual bound increases over time (See Figure 2). Also, all primal heuristics will be deactivated, so that the focus is only on proving optimality via branching.

**Environment:** Environment is a traditional branch-and-bound algorithm. The solver stops after each node is processed (LP solved), receives a branching decision, performs branching, and selects the next open node to process. The solver remains in the SOLVING stage until the episode terminates (time limit of 15 minutes reached, or when the problem is solved). Action is one of the current node’s branching candidates (non-fixed integer variables). Only single-variable branching is allowed.

**Dual integral:** This objective measures the area over the curve of the solver’s dual bound (a.k.a. global lower bound), which usually corresponds to a solution of a valid relaxation of the MILP. By branching, the LP relaxations corresponding to the branch-and-bound tree leaves get tightened, and the dual bound increases over time. With a time limit  $T$ , the dual integral expresses as:

$$Tc^\top x^* - \int_{t=0}^T z_t^* dt \quad (2)$$

where  $z_t^*$  is the best dual bound at time  $t$ , and  $Tc^\top x^*$  is an instance-specific constant that depends on the optimal solution value  $c^\top x^*$ . The dual integral is to be minimized and takes an optimal value of 0.

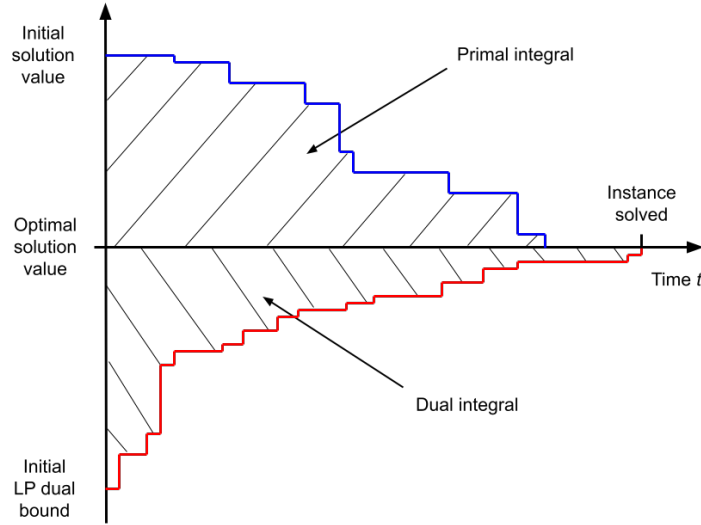


Figure 2: Primal and dual bounds evolution vs solving time. Image from [10]

**Datasets:** There are three problem benchmarks from diverse application areas. A problem benchmark consists in a collection of mixed-integer linear program (MILP) instances in the standard MPS file format. The first two problem benchmarks are inspired by real-life applications of large-scale systems at Google, while the third benchmark is an anonymous problem inspired by a real-world, large-scale industrial application [10].

- Problem benchmark 1: Balanced Item Placement. This problem deals with spreading items (e.g., files or processes) across containers (e.g., disks or machines) utilizing them evenly. Items can have multiple copies, but at most, one copy can be placed in a single bin. The number of items that can be moved is constrained, modeling the real-life situation of a live system for which some placement already exists. Each problem instance is modeled as a MILP, using a multi-dimensional multi-knapsack formulation. This dataset contains 10000 training instances (pre-split into 9900 train and 100 valid instances).
- Problem benchmark 2: Workload Apportionment. This problem deals with apportioning workloads (e.g., data streams) across as few workers (e.g., servers) as possible. The apportionment is required to be robust to any one worker’s failure. Each instance problem is modeled as a MILP, using a bin-packing with apportionment formulation. This dataset contains 10000 training instances (pre-split into 9900 train and 100 valid instances).
- Problem benchmark 3: Anonymous Problem. The MILP instances corresponding to this benchmark are assembled from a public dataset, whose origin is kept secret to prevent cheating. Reverse-engineering for the purpose of recovering the test set is explicitly forbidden. This dataset contains 118 training instances (pre-split into 98 train and 20 valid instances).

## 5 Our Solution

In this section, we first provide an overview of the Graph Convolutional Neural Network approach adopted from [13], then we provide results of our experiments, ablations, and insights around them.

### 5.1 Graph Convolutional Neural Network (GCNN)

The GCNN approach introduced in [13] adopts an imitation learning approach to learn a fast approximation of strong branching (SB). This method models the branching policy as a graph convolutional neural network (GCNN), that in turn allows to leverage the bipartite graph representation of MILP problems. In this setting, a classification model is trained to predict the branching variable selection of the SB agent, hence the training loss function can be the softmax with cross-entropy.

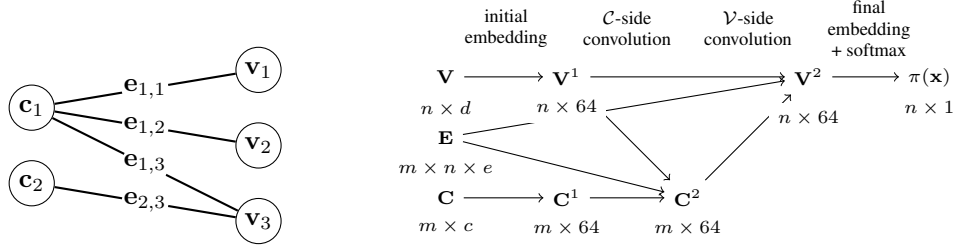


Figure 3: Left: bipartite state representation  $s_t = (\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$  with  $n = 3$  variables and  $m = 2$  constraints. Right: bipartite GCNN architecture for parametrizing the imitation policy  $\pi$ . Visualization from [13].

Figure 3 shows the bipartite representation model adopted in the GCNN approach. In this figure, node and edge features are denoted by  $(\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$ . Moreover,  $\mathbf{C} \in \mathbb{R}^{m \times c}$ ,  $\mathbf{V} \in \mathbb{R}^{n \times d}$ ,  $\mathbf{E} \in \mathbb{R}^{m \times n \times e}$  respectively refer to constraints features (one per row), variable features (one per column), and edge features (sparse tensor). Table 1 further shows the details of the features incorporated.

The GCNN approach utilizes two interleaved convolutions, one from variables to constraints and one from constraints to variables. In addition, the neural network used incorporates relu activations as well as prenorm layers.

Table 1: Constraints, edges, and variables features used in  $s_t = (\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$ ; adopted from [13].

Tensor	Feature	Description	
$\mathbf{C}$	obj_cos_sim	Cosine similarity with objective.	
	bias	Bias value, normalized with constraint coefficients.	
	is_tight	Tightness indicator in LP solution.	
	dualsol_val	Dual solution value, normalized.	
$\mathbf{E}$	age	LP age, normalized with total number of LPs.	
	coef	Constraint coefficient, normalized per constraint.	
	type	Type (binary, integer, impl. integer, continuous) as a one-hot encoding.	
	coef	Objective coefficient, normalized.	
	has_lb	Lower bound indicator.	
	has_ub	Upper bound indicator.	
	$\mathbf{V}$	sol_is_at_lb	Solution value equals lower bound.
		sol_is_at_ub	Solution value equals upper bound.
		sol_frac	Solution value fractionality.
		basis_status	Simplex basis status (lower, basic, upper, zero) as a one-hot encoding.
reduced_cost		Reduced cost, normalized.	
age		LP age, normalized.	
sol_val		Solution value.	
inc_val		Value in incumbent.	
avg_inc_val	Average value in incumbents.		

## 5.2 Results

### 5.2.1 Results of variable selection (classification accuracy)

Table 2 shows the accuracy of the classification task over a number of different experiments on the item-placement dataset. In each one of these experiments, we generated a training dataset by recording the states of the MILP branching and forming a classification problem from the variable selection. Therefore, one can use different environment settings to generate different training samples set. These settings may include: time-limit to solve or terminate (in training), probability of strong branching (to use SB only on a portion of samples and pseudo-cost on the rest), number of training samples recorded (we can keep generating samples until a desired count is reached), etc.

We can see in Table 2 that in general models with higher top-k classification accuracy are somehow correlated with higher overall reward (better dual integral hence convergence), however, this relationship is not linear and thus better top-1 does not guarantee a better reward. Moreover, it is worth noting that the load-balancing and anonymous datasets also demonstrate a similar behavior, further reinforcing the conclusion made (results omitted for brevity).

**Remark #1:** Top-1 accuracy is not enough. One needs to always check the final reward over the evaluation set (validation or test). It is also worth noting that in addition to the reward evaluated here (i.e. dual integral), there are other classical benchmarking metrics such as: “1) shifted geometric mean of the solving times in seconds, including running times of unsolved instances without extra penalization (Time); 2) the hardware-independent final node counts of instances that are solved by all baselines (Nodes); and 3) the number of times each branching policy results in the fastest solving time, over the number of instances solved (Win)” [13].

Table 2: Classification accuracy for various experiments on the item-placement validation set.

Method	Time-Limit	P of SB	Top-1 (%)	Avg. Accumulated Reward
FSB	15 min	1.0	-	3771
Reliable Branching	15 min	-	-	3506
GCNN	1 min	1.0	93.9	6663
GCNN	15 min	1.0	98.6	6638
GCNN	60 min	1.0	98.9	6669
GCNN	600 min	1.0	99.3	6449
GCNN*	15 min	0.001	67.2	7221

\* Refers to our best result, which will be explained later.

### 5.2.2 GCNN results on the validation set

Table 3 shows the results of our best trained models over the validation set problems of the three datasets. The evaluation time-limit is 15 minutes and the same setting as the test evaluations are used. Since there are 100 MILPs in the evaluation set, it takes around 24 hours for a complete evaluation on the validation set MILP instances (for the Anonymous dataset there are only 20 validation MILPs but as outlined in the competition we run them 5 times with different seeds).

For training, we used a batch size of 64 for the Item-Placement and Anonymous datasets and a batch of size 32 for the Load-Balancing (since it has larger problems that occupy more GPU memory). We trained on servers with  $8 \times V100$  GPUs with 32GB memory, for 50 epochs. We utilized an Adam optimizer with learning rate of 0.001, decay on plateau (10 epochs) and early termination of 20 epochs of non-decreasing loss. Training samples set size is 100K for Item-Placement and Anonymous datasets, but only 20K for the Load-Balancing dataset as data collection is slower for this dataset.

Table 3: GCNN results on the validation set MILPs.

Dataset	Data Size (Train-Val)	Embedding Dimension	Time Limit	P of SB	Top-1 (%)	Reward
Item-Placement	100K-1K	8	15 min	0.001	67.2	7221
Load-Balancing	20K-1K	64	1 min	0.01	8.0	625363
Anonymous	100K-1K	64	15 min	1.0	41.6	32517856

### 5.2.3 Full test set evaluation results

Table 4-6 show the final test results of the competition, among all the submissions for the dual task. Our submission is “EI-OROAS”. The models used for generating Table 3 were used in this submission. It can be observed that the top two submissions are very close. GCNN achieves top rank on the Load-Balancing dataset and rank 2 with a small margin on the Item-Placement dataset. It is also worth noting that after the competition deadline, we were able to achieve even better results on these two datasets. Also, in Table 4 we only show the top 10 best results, out of the overall 23 teams [10]. Table 7 shows the average ranks over the three datasets.

**Remark #2:** GCNN results are strong. This means even though there have been papers published after the original GCNN paper [13], proper tuning can still keep GCNN performance on top.

**Remark #3:** An interesting observation is that there are 16 different teams in the top 10 rewards of three datasets. This means unique team solutions were not able to achieve equally good results on different datasets, suggesting data-dependent solutions may on average be the way to go. That being said, our GCNN results are strong across all datasets, proving GCNNs are promising for these tasks.

Table 4: Item-Placement top 10 test rewards from the competition website [10].

Team Name	Reward	Rank
Nuri	6684.0	1
<b>EI-OROAS</b>	<b>6670.3</b>	<b>2</b>
EFPP	6487.53	3
lxj24	6443.55	4
ark	6419.91	5
qqy	6377.23	6
KAIST_OSI	6196.56	7
nf-lzg	6077.72	8
Superfly	6024.20	9
Monkey	5978.65	10

Table 5: Load-Balancing top 10 test rewards from the competition website [10].

Team Name	Reward	Rank
<b>EI-OROAS</b>	<b>631744.31</b>	<b>1</b>
KAIST_OSI	631410.58	2
EFPP	631365.02	3
DaShun	630898.25	4
blueterrier	630826.33	5
Nuri	630787.18	6
gentlemenML4CO	630752.94	7
comeon	630750.66	8
Superfly	630746.96	9
Monkey	630737.85	10

Table 6: Anonymous top 10 test rewards from the competition website [10].

Team Name	Reward	Rank
Nuri	27810782.42	1
qqy	27221499.03	2
null_	27184089.51	3
<b>EI-OROAS</b>	<b>27158442.74</b>	<b>4</b>
DaShun	27151426.15	5
KEP-UNIST	27085394.46	6
lxj24	27052321.48	7
THUML-RL	26824014.00	8
KAIST_OSI	26626410.86	9
Superfly	26373350.99	10

Table 7: Average ranks over the three datasets for top 10 teams in the ML4CO competition [10].

Team Name	Rank Multiplication	Average Rank
<b>EI-OROAS</b>	<b>8</b>	<b>2.33</b>
Nuri	6	2.67
KAIST_OSI	126	6
EFPP	117	6.33
qqy	132	6.33
DaShun	280	7.67
Superfly	810	9.3
lxj24	532	10
comeon	1056	10.3
Monkey	1100	10.3

#### 5.2.4 Partial test set evaluation results prior to the final evaluation

Throughout the ML4CO competition, leaderboard was updated on a weekly basis. GCNN results were consistently strong throughout the competition timeline. Table 8 shows the last 4 weeks of leaderboard updates for top 5 teams. Note that the evaluations were only on 25% of the final test set.

Table 8: Average ranks of the last 4 leaderboard updates over the three datasets.

Team Name	Oct.04	Oct.08	Oct.15	Oct.23	Overall Average Rank
<b>EI-OROAS</b>	<b>1</b>	<b>1.33</b>	<b>2.67</b>	<b>1.67</b>	<b>1.67</b>
Nuri	2.33	4.33	3.67	3.67	3.5
KAIST_OSI	8.67	3.67	6.67	4.67	5.92
EFPP	-	-	-	5	5
qqy	5.67	4.67	6.67	8	6.25

#### 5.2.5 Inference speed and latency

Capacity of a neural network is generally related to its number of parameters. Bigger models with a higher number of parameters usually lead to higher degrees of performance. For our problem of variable selection however, choosing very large architectures will be counter-productive, as these larger models come at a cost of slower inference. This means each classification within each solver iteration is slower. Sometimes this may still lead to good reward values if being accurate is more important; sometimes on the other hand speed may become a more important factor and we may be able to tolerate a certain degree of classification error. This will also be data-dependent. Smaller size problems may get solved faster, so it won't make sense to use a large NN in this setting. For harder/larger MILP instances however, it may be beneficial to choose a bigger model to push the accuracy up. Table 9 shows the size and inference latency (averaged over 10000 runs) of the models used in our final submission. As we can see from this table, Item-Placement dataset has clearly benefited from a smaller swift model.

**Remark #4:** We need to carefully find a balance between NN accuracy and speed. Smaller models tend to be less accurate on classification (variable selection), but because they are faster, overall they lead to better solutions (closing the dual gap faster, as long as they can learn), hence higher rewards.

Table 9: DNN model size and inference latency (per node in milli-seconds on validation MILPs).

Dataset	Embedding Dimension	Model Size	# parameters	Inference Latency	Top-1 (%)	Reward
Item-Placement	8	20 KB	1264	0.12	67.2	7221
Load-Balancing	64	270 KB	63872	9.25	8.0	625363
Anonymous	64	270 KB	63872	0.74	41.6	32517856



### 5.2.6 Training and evaluation time

Table 10 shows the time that takes to perform training for 50 epochs and evaluation on the 100 validation set MILPs on an 8-node V100 server. We can see that training is relatively fast, but evaluation takes much longer due to the time-limit of 15 minutes that is used in the competition.

Table 10: Training and evaluation time (on validation set MILPs with 15 minutes solver time-limit)

Dataset	# Training MILPs	# Training Samples	Training Time (hr)	# Evaluation MILPs	Evaluation Time (hr)
Item-Placement	9.9K	100K	≈ 0.6	100	≈ 25
Load-Balancing	9.9K	20K	≈ 1.2	100	≈ 25
Anonymous	100	100K	≈ 0.84	20	≈ 25

### 5.2.7 Significant variation in performance when using a same architecture but collecting different training samples

Figure 4, 5, and 6 show scatter plots of validation MILPs reward versus the solving time-limit, data size, and probability of strong branching, over a large number of experiments.

**Remark #5:** As observed, there is a significant variation in the validation reward when different flavours of training samples are collected. This means one needs to investigate various kinds of training samples settings in order to obtain the best policy.

**Remark #6:** Since we can collect different sets of training samples from a same set of MILP instances, a fair comparison of different methods is only possible if the original MILP instances are provided. This is the case in the ML4CO challenge. However, for problem instances such as Set Covering, Combinatorial Auction, Capacitated Facility Location, or Maximum Independent Set type problems, that are largely used in the research papers, often times it is not the case. We argue that generating different training sample sets from say Set Covering problems can lead to different final reward values. Therefore, we recommend to use a standard benchmark of MILP instances that are publicly released for such kinds of performance evaluations (e.g. datasets used in the ML4CO challenge).

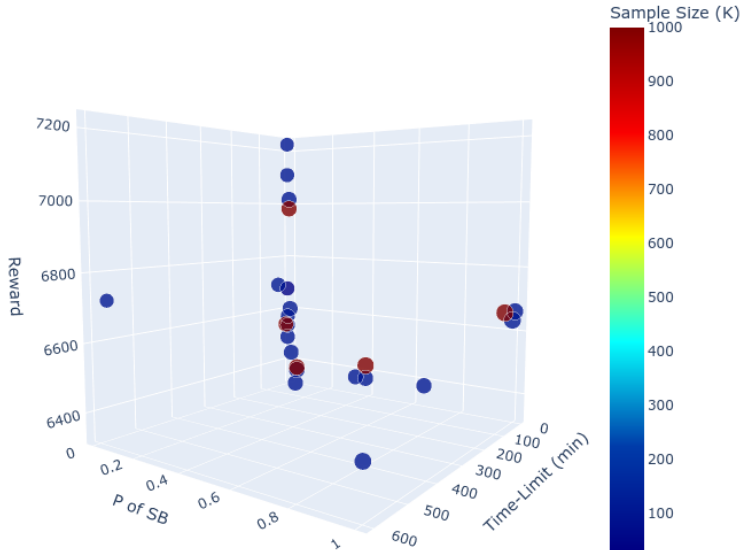


Figure 4: Validation reward in terms of solving time-limit, probability of strong branching, and training sample size used in the data collection process for different experiments (item-placement).

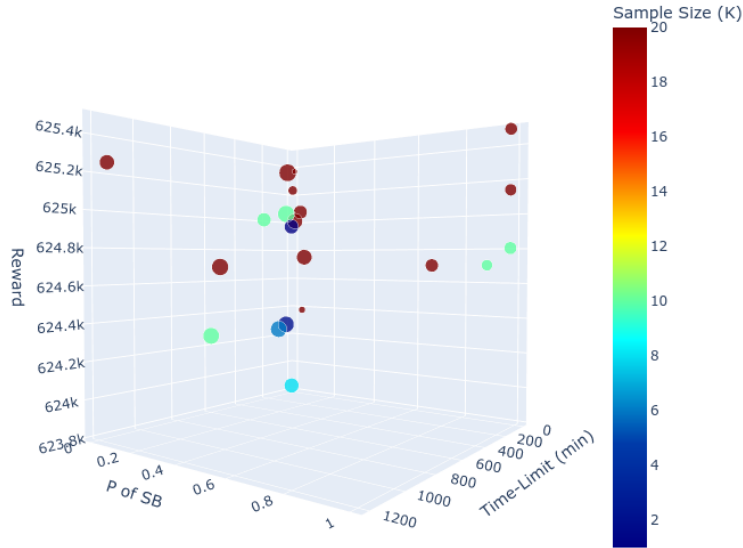


Figure 5: Validation reward in terms of solving time-limit, probability of strong branching, and training sample size used in the data collection process for different experiments (load-balancing).

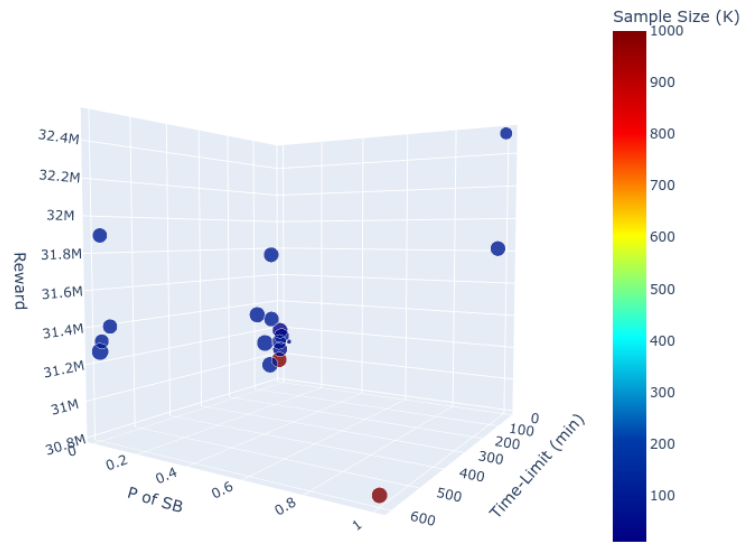


Figure 6: Validation reward in terms of solving time-limit, probability of strong branching, and training sample size used in the data collection process for different experiments (anonymous).

### 5.2.8 The impact of the training sample set size

Figure 4, 5, and 6 show the impact of training sample set size in the overall reward for the three datasets (see the color coding).

**Remark #7:** We observe that larger training set size does not necessarily result in a higher reward. This is unlike the common intuition that more training data can reduce the over-fitting.

### 5.2.9 Top-1, Top-3, Top-5, and Top-10

As mentioned previously in this section, top-1 variable selection accuracy does not correlate well with the overall reward. For example, a large model with high capacity, will likely achieve a high top-1 classification accuracy, but if it gets too slow, it may end up hurting the final reward as it can do

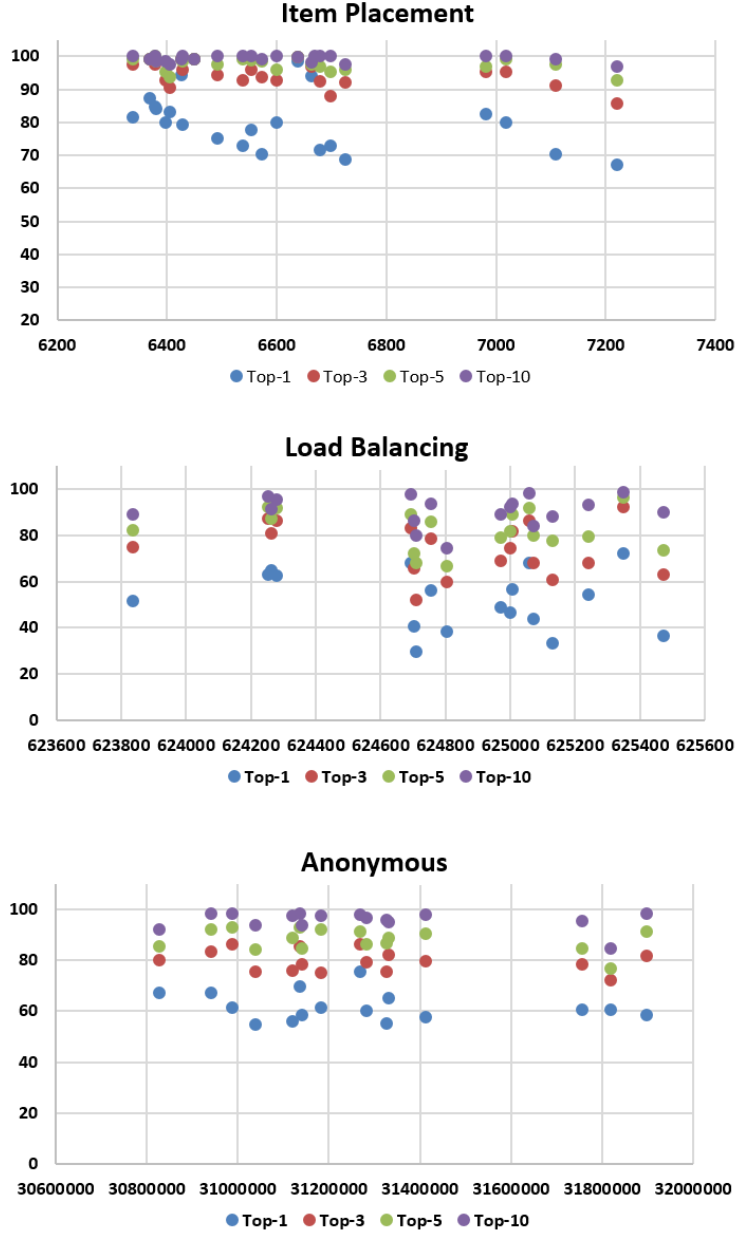


Figure 7: Validation top-1, top-3, top-5, and top-10 versus the overall reward.

fewer number of iterations within a given time. Figure 7 demonstrates the reward-top-k scatter plot for the validation sets over a large number of experiments. We observe that:

**Remark #8:** Top-5 and top-10 variable selection is in general very accurate, most of the times above 90%. This suggests that variable selection in the branch-and-bound tree is in general fairly accurate with GCNNs, thus overall they show a strong performance compared to conventional heuristic rules.

### 5.2.10 Data-Centric AI for CO

A trend has recently been emerging in the AI community to design data-centric AI systems and tools, as opposed to only model-centric approaches [24]. The idea is that the community has focused highly on building new algorithms and models, however, in practice data drift results in considerable mismatch between the training set and test/deployment environment. Based on the

results of our experiments, we also conclude that data engineering plays an important role in the combinatorial optimization learning approaches, and the community needs to pay equal attention to both model/algorithm improvements and data engineering efforts.

### 5.2.11 Other thoughts

There are other ideas that we didn't try due to lack of time and resources, including: a) applying data augmentation, and b) employing multiple models. In terms of augmentation, it is important to note that one needs to take into account the nature of the data that is on linear equations when designing the augmentation operations. As for using multiple models, this idea comes from the fact that when we applied a clustering algorithm on the anonymous dataset, it revealed 3 cluster of problems, suggesting that training 3 separate GCNN models, one for each cluster of problems, may lead to performance improvements. We leave these ideas for the interested readers to explore.

## 6 Conclusion

This paper summarizes the solution and lessons learned by the Huawei EI-OROAS team in the dual task of the NeurIPS 2021 ML4CO competition. The submission of our team achieved the second place in the final ranking, with a very close distance to the first spot. In addition, our solution was ranked first consistently for several weekly leaderboard updates before the final evaluation. We provide insights gained from a large number of experiments, and argue that a simple Graph Convolutional Neural Network (GCNNs) can achieve state-of-the-art results if trained and tuned properly.

## References

- [1] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.
- [2] Tobias Achterberg and Timo Berthold. Hybrid branching. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 309–311, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-01929-6.
- [3] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005. ISSN 0167-6377. doi: <https://doi.org/10.1016/j.orl.2004.04.002>. URL <https://www.sciencedirect.com/science/article/pii/S0167637704000501>.
- [4] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A Machine Learning-Based Approximation of Strong Branching. *INFORMS Journal on Computing*, 29(1):185–195, February 2017. doi: 10.1287/ijoc.2016.0723.
- [5] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Finding cuts in the tsp. *Technical report*, 1995.
- [6] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d’horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.
- [7] Bob Bixby. The gurobi optimizer. *Transp. Re-search Part B*, 41(2):159–178, 2007.
- [8] Christian Blum, Jakob Puchinger, Günther R Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied soft computing*, 11(6):4135–4151, 2011.
- [9] ecole.ai. Ecole: a library for learning inside the scip solver. <https://www.ecole.ai/>, .
- [10] ecole.ai. ML4CO: 2021 neurips competition on machine learning for combinatorial optimization. <https://www.ecole.ai/2021/ml4co-competition/>, .
- [11] Matthias Ehrgott and Xavier Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-spektrum*, 22(4):425–460, 2000.
- [12] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eiffler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, et al. The scip optimization suite 7.0. 2020.
- [13] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems 32*, 2019.

- [14] Prateek Gupta, Maxime Gasse, Elias B Khalil, M Pawan Kumar, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. In *Advances in Neural Information Processing Systems 33*, 2020.
- [15] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.
- [16] Clemens Heuberger. Inverse combinatorial optimization: A survey on problems, methods, and results. *Journal of combinatorial optimization*, 8(3):329–361, 2004.
- [17] Zeren Huang, Kerong Wang, Furui Liu, Hui-Ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition*, 123:108353, 2022.
- [18] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.
- [19] Elias B. Khalil, Andrea Lodi, Didier Chételat, Antoine Prouvost, Bistra Dilkina, Maxime Gasse, Zarpellon Giulia, and Lauren Charlin. Recent advances in integrating machine learning and combinatorial optimization. *AAAI*, 2021.
- [20] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. ISSN 00129682, 14680262. URL <http://www.jstor.org/stable/1910129>.
- [21] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.
- [22] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O’Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.
- [23] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O’Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.
- [24] Andrew Ng, Lora Aroyo, et al. Neurips data-centric ai workshop. <https://datacentricai.org/>, 2021.
- [25] Jagat Patel and John W. Chinneck. Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Math. Program.*, 110(3):445–474, 2007. ISSN 0025-5610.
- [26] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3931–3939, 2021.