ML4CO submission EFPP

Edward Lam* **Frits de Nijs** **Pierre Le Bodic** **Peter J. Stuckey**

Faculty of IT, Monash University
{Edward.Lam, Frits.Nijs, Pierre.LeBodic, Peter.Stuckey}@monash.edu

Abstract

This paper presents our submission to ML4CO, a branching rule that exploits the structure within a problem class via a graph neural network that is trained using a distributional reinforcement learning algorithm. The learned branching rule participated in the dual bound challenge, ranking third among 23 entries. Experiments on the ML4CO instances show that it performs between 16% and 34% better on average than the default reliability pseudocost branching in the state-of-the-art academic solver SCIP.

1 Introduction

This paper presents our submission to dual bound challenge in the ML4CO competition. Our proposal ranked third overall in the final evaluation. We developed a branching rule based on the state-of-the-art Fully-parameterized Quantile Function (FQF) distributional reinforcement learning algorithm [5]. We implemented four main novelties: 1) The branching rule is trained in an environment that is arguably easier to learn and is different to the test environment. 2) The branching rule contains a graph neural network that is trained on an encoding of the problem rather than of the variables and constraints of a Mixed-Integer Programming (MIP) model. 3) The results of calling the neural network is cached and recalled at alternating depths of the search tree to avoid expensive calculations. 4) The probability distribution of rewards learned by FQF is constrained to be monotonically non-increasing.

2 Solution Methodology

Agent architecture We base our agent architecture on the Fully-parameterized Quantile Function (FQF) reinforcement learning (RL) algorithm. FQF is a state-of-the-art *distributional* RL algorithm [5], a class of algorithms which attempt to learn the full probability distribution $Z^\pi = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$ over returns (γ-discounted sum of rewards $R(s, a)$ per chosen state-action), as opposed to only its expected value $Q^\pi = \mathbb{E}[Z^\pi]$ [2]. The FQF algorithm operates on the inverse cumulative distribution function of the returns, $F^{-1}_{Z^\pi(s,a)}(p)$, which is then approximated by a staircase function supported on n quantile fractions. Two neural networks are used to capture this staircase function: the *fraction proposal network* which learns to predict the optimal position of the support (quantile cut points) τ_i, and the *quantile value network* which learns to predict the value of the return $F^{-1}_{Z^\pi(s,a)}(\tau_i)$ at the position of a support.

To extract features from the solver state, we replace the typical Convolutional Neural Network with a Graph Neural Network (GNN) architecture. GNNs are layers in end-to-end deep learning models for processing graph-structured data and enable learning on data with complex relationships. GNNs take numbers (i.e., the features) from each vertex and apply a function before accumulating this number with the numbers from its neighbors. We employ the Graph Isomorphism Network (GIN), which is theoretically shown to be more powerful than earlier GNNs as it learns the same representation for

*Contact Author

NeurIPS 2021 Competition - Machine Learning for Combinatorial Optimization (ML4CO).

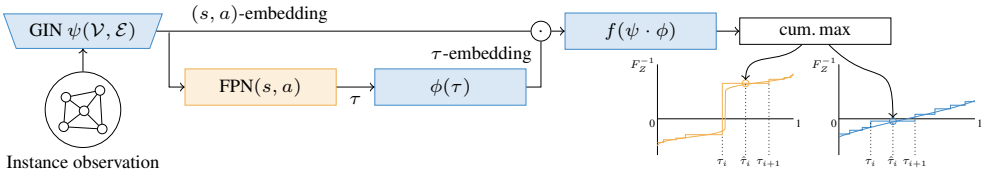


Figure 1: The neural architecture. Colored boxes contain trainable parameters. The blue elements are trained against value loss and the orange FPN trained against fraction loss.

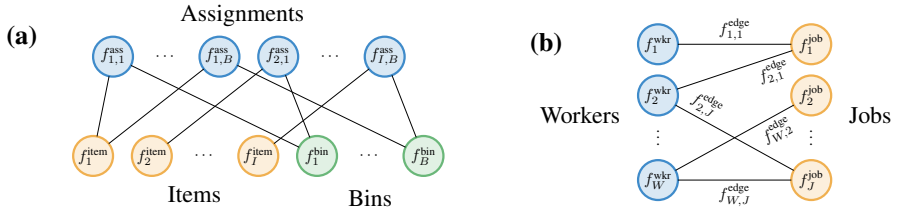


Figure 2: Compact graph encodings of the entities in (a) item placement, (b) load balancing instances.

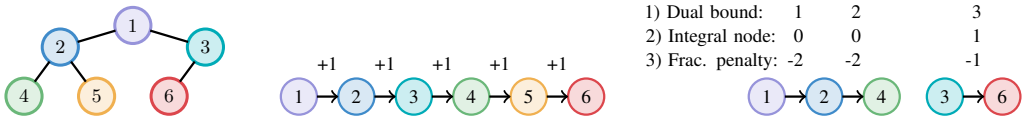
isomorphic graphs [4]. As we have edge features, we compute feature values according to the model of Hu et al. [3]. For a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V}$ and edges $\mathcal{E}$, the output features f_i' of a vertex $i \in \mathcal{V}$ are computed as $f_i' = M\big(f_i + \sum_{j \in \mathcal{N}(i)} \mathrm{ReLU}(f_j + L(f_{j,i}))\big)$, where $\mathcal{N}(i) = \{j : (i, j) \in \mathcal{E}\}$ are the neighbors of i, and $f_{j,i}$ are the edge features. Trainable functions M and L are respectively: a multi-layer perceptron, and a linear layer that transforms the dimensions of $f_{j,i}$ to those of f_j.

The published FQF algorithm places no constraints to ensure that the output of the quantile value network is always non-decreasing. We additionally constrain the distribution to be monotonic by projecting the predicted returns per action through a cumulative maximum (cum. max) operation, such that $F^{-1}_{Z^\pi(s,a)} = \text{cum. max}_{\tau_i} F^{-1}_{Z^\pi(s,a)}(\tau_i)$. Imposing monotonicity improved performance on a related distributional algorithm [6]. The entire neural network is summarized in Figure 1.

Observations We use a custom observation function to extract salient features. For every variable x, define its *dynamic features* $F(x) = (F^1(x), \ldots, F^{11}(x))$ as a vector containing values extracted from the state of the solver. $F^1(x) = x$ is the value of the variable. $F^2(x) = |\lceil x \rceil - 0.5 - x|$ is the fractionality of the variable. $F^3(x) = 1$ if x is fractional and 0 otherwise. $F^4(x)$ is the *reduced cost* of x. $F^5(x)$ and $F^6(x)$ are the down and up pseudocost of x, $F^7(x)$ and $F^8(x)$ are the counts of down or up branches on x, respectively. These represent how reliable the pseudocosts are. $F^9(x), F^{10}(x), F^{11}(x)$ contain a one-hot encoding of the basis status of x, having value 1 if at lower bound, basic, or at upper bound respectively.

Typically, features of the variables and constraints of a MIP are encoded in what we call the *MIP graph*. The MIP graph is bipartite with vertices representing variables on one side and constraints on the other, each with their relevant features. This encoding is applicable to any MIP problem but its drawback is its size: it has one vertex for every variable and constraint. For the known instances, we develop compact encodings containing only features of the objects in the problem definition (i.e., items, bins, workers), instead of features of the MIP model. The smaller representation has several advantages over the MIP graph. Firstly, because the neural network is relieved from learning the constraints explicitly and instead learns a compressed approximation of the constraints, a smaller size becomes possible, ultimately leading to faster training. Secondly, this direct approach eases the need to learn a MIP solver. For example, different bases (selections of independent columns in the model, represented through features 9–11) can correspond to the same solution, which would have to be learned if learning on the formulation. This is clearly a monumental task. Thirdly, as researchers better understand the problem and develop improved models after the competition ends, this encoding remains valid as it is architecture independent. The graph encoding the instance data and the solver state for the two problem classes are presented in Figure 2.

Transition model The *original MDP* defined by the competition environment presents a partially observable view into the MIP solver, where each state observation corresponds to the current node being solved. As MIP solvers typically use a best-first node selection rule to determine the next node to branch in, the next state is frequently not a child of the current node, making the original transition



(a) Branch-and-bound tree. (b) Original MDP for (a), with reward following global bound change. (c) Artificial MDPs for (a), together with our three proposed reward functions.

Figure 3: An example of the artificial MDPs derived from the original MDP. The number within each node is its dual bound and, if using best-first node selection, it is also the ordering of the solved nodes. The original MDP follows this ordering until all nodes encountered, rewarding the agent with the change to global dual bound per step, which together with the solve time defines the dual integral reward function used to rank entrants in the competition.

function a complex interaction between the agent-controlled branching rule and the solver-controlled node selection rule. See for example nodes 5 and 6 in Figure 3a, which are far apart in the branch-and-bound tree, but consecutive states in the original MDP. To make the impact of decisions more directly related to the branching outcome, we record training experiences as if they came from an artificial MDP (Figure 3c) in which the next state corresponds to the child node with the worst (lowest) dual bound. When a node has no further children, this node is considered to terminate the artificial episode. Experiences from this artificial MDP are constructed by transforming entire episodes drawn from original MDP; we do not change the environment itself.

Reward Functions We test three reward functions (RFs) for this artificial MDP, as shown in Figure 3c: 1) *Dual bound RF* is the difference between the dual bound of the current node and the worst dual bound of its two children. This reward signal drives the agent to produce child nodes with higher dual bounds. 2) *Integral node RF* rewards the agent with value 1 for taking actions leading to integral or infeasible nodes (i.e., leaves). 3) *Fractional node penalty RF* penalizes each fractional child by -1, resulting in reward 0 only if both children are leaves. Both 2) and 3) lead the agent to generate small subtrees, indirectly improving the global dual bound. Eventually, half the nodes (the leaves) of the branch-and-bound tree are integral or infeasible, resulting in high rewards. However, at the start the majority of nodes are fractional, resulting in a sparse reward function that may be difficult to learn, especially when solving time limits are imposed. Preliminary experiments indicated 3) performs best and hence it is used in our competition entry.

Exploration Strategies While ϵ-greedy exploration effectively solves RL problems with simple action spaces, it is unlikely to be sufficient in learning to branch because of the huge action set of fractional variables. However, FQF is an off-policy algorithm, meaning that it can learn from experiences that do not match the current policy's choices. We exploit this fact by using random branching and reliability pseudocost branching to generate examples to fill the replay buffer, in addition to the typical ϵ-greedy exploration strategy. Upon starting a new episode (i.e., solving a new instance), one of these three exploration strategies is chosen uniformly at random.

Efficient Policy Execution The forward pass of the neural network can be a time-consuming operation. Preliminary experiments suggest that caching the results of calling the neural network and reusing these outputs at alternating depths in the branch-and-bound tree achieves better performance.

3 Experimental Evaluation and Conclusions

Our training code is implemented using PyTorch 1.9.1 and the GNN library from PyTorch Geometric 1.7.2. The training code uses 1 training thread, 30 threads for exploring the environment and generating training data, and 20 threads for scoring the latest policy against the incumbent policy on a few evaluation instances. We trained the algorithm for four weeks for the competition, on a 64-core, 8 A100 GPU Nvidia DGX server; we used the competition-suggested training/validation split.

Comparison of Branching Rules Table 1a compares the dual integral and the optimality gap, averaged over the test instances, of the Learned branching rule against the state of the art (Reliability) and the naïve baselines Random and Most Fractional branching rules.

The Learned branching rule clearly outperforms the state of the art for the Item Placement problem. The ranking of the three established algorithms also corroborate well-known results [1]. For the load

Table 1: Average dual integral and optimality gap of a) branching rule with percentage difference to reliability branching, and b) configuration with percentage difference to the full algorithm. All results are calculated using the same best-known primal bound provided by the competition organizers.

a) evaluation	Item Placement		Load Balancing	
	Dual Integral	Gap	Dual Integral	Gap
Reliability	4025	55%	6923	1.2%
Most Fractional	6049 (+50%)	76% (+37%)	6158 (-11%)	0.9% (-21%)
Random	6083 (+51%)	77% (+38%)	6197 (-10%)	0.9% (-22%)
Learned	2643 (-34%)	44% (-21%)	5823 (-16%)	0.8% (-30%)

b) ablation	Item Placement		Load Balancing	
	Dual Integral	Gap	Dual Integral	Gap
Full	3324	51%	5660	0.8%
Original MDP	4812 (+45%)	65% (+27%)	6023 (+6%)	0.8% (+10%)
Dual bound RF	4519 (+36%)	61% (+20%)	5906 (+4%)	0.8% (+3%)
Integral node RF	3602 (+8%)	53% (+4%)	6091 (+8%)	0.8% (+10%)
Call every depth	3677 (+11%)	53% (+3%)	5778 (+2%)	0.8% (+4%)
MIP graph	3995 (+20%)	55% (+8%)	5856 (+3%)	1.1% (+49%)
Non-monotonic FQF	3632 (+9%)	53% (+5%)	5839 (+3%)	0.8% (+8%)

balancing problem, Reliability is outperformed by Random, itself outperformed by Learned, which suggests that this problem exhibits properties not correctly exploited by the state of the art.

Ablation Study Table 1b shows the impact of removing individual components from the full training and execution algorithm. For this ablation study, every variant is trained for one week. These results indicate that all components contribute to the performance of the learned branching rule. Training on the original MDP or using the full MIP graph observation degrades performance most significantly, the second observation potentially explaining the large discrepancy between our position on Anonymous (13^{th}) and the other domains (3^{rd}). The two reward functions that drive the agent towards smaller branch-and-bound trees (the integral node RF and the fractional node penalty RF in Full) also have a large impact on performance.

Conclusions This paper showed that a data-driven branching rule trained on instances from a single problem class can outperform the state-of-the-art reliability pseudocost branching in MIP. However, achieving this result required exploiting expert knowledge in MIP to customize the reinforcement learning problem, rather than blindly training in the given environment.

Many avenues for future research directions are available. Due to the time constraints imposed by the ML4CO competition, many of the design choices are chosen to be simple rather than high performing. Many of the hyperparameters, including neural network sizes and the number of supports in the approximated distribution of FQF, are arbitrarily chosen without tuning. Epsilon-greedy exploration likely cannot cope with the large action space, requiring a more sophisticated exploration strategy.

References

[1] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42 – 54, 2005.

[2] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *34th Intl. Conf. on Machine Learning*, pages 693–711, 2017.

[3] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *Intl. Conf. on Learning Representations*, 2020.

[4] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *Intl. Conf. on Learning Representations (ICLR)*, 2019.

[5] Derek Yang, Li Zhao, Zichuan Lin, Tao Qin, Jiang Bian, and Tie-Yan Liu. Fully parameterized quantile function for distributional reinforcement learning. In *Neural Information Processing Systems 32*, 2019.

[6] Fan Zhou, Zhoufan Zhu, Qi Kuang, and Liwen Zhang. Non-decreasing quantile function network with efficient exploration for distributional reinforcement learning. In *Intl. Joint Conf. on Artificial Intelligence*, pages 3455–3461, 2021.