Accelerating Linear Programming Solving by Exploiting the Performance Variability via Reinforcement Learning

Xijun Li[1,2], **Qingyu Qu**[3], **Fangzhou Zhu**[2], **Mingxuan Yuan**[2], **Jia Zeng**[2], **and Jie Wang**[1]

[1]University of Science and Technology of China
[2]Huawei Noah's Ark Lab
[3]Beihang University

Abstract

The trend of using machine learning techniques to improve the mathematical programming solvers has recently drawn lots of attention. Most previous work focuses on replacing key components within the solvers using machine learning techniques. Empirically, practitioners observed that the solving efficiency of the solver is highly sensitive to the formulation of inputted mathematical programming models, such as the appearing order of variables (one of performance variability of solvers). In other words, an inappropriate formulation might harm the performance robustness of the solver. Instead, we exploit this type of performance variability, proposing a novel approach for accelerating linear programming solving via reinforcement learning-based reformulation. We implemented the proposed approach with three reputable solvers, i.e., Gurobi, SCIP, and CLP. We conducted extensive experiments over two public LP datasets from NeurIPS 2021 ML4CO competition and one large-scale LP dataset collected from a real-life production planning scenario. Experimental results demonstrate that the proposed approach effectively reduces the solving iteration number (20%↓ on average) and solving time (15%↓ on average) over the above datasets, compared to directly solving the original linear programming models. This work can inspire the future research for better exploiting the performance variability of solvers with machine learning techniques.

1 Introduction

Through many years of practice, it has been verified that mathematical programming (Ge et al. 2021) is capable of formulating real-life optimization problems such as planning, scheduling, resource allocation, etc. The mathematical programming model can obtain the optimal solution by resorting to corresponding solvers, such as Gurobi (Gurobi 2021), CPLEX (IBM 2021), SCIP (ZIB 2021), COIN LP (CLP) (COIN-OR Foundation 2021a), etc. Government and business corporations benefit significantly from the practice of mathematical programming in their daily operations (Mavrotas and Makryvelios 2021), which thus constantly draws interests from academics and industries. There are many kinds of mathematical programmings, including linear programming (LP), mixed integer programming (MIP), and quadratic programming (QP). In past decades, a collection of classical algorithms (such as simplex (Dantzig 1987), barrier (Andersen, Roos, and Terlaky 2003), branch and bound (Wolsey 2007), etc.) have been proposed to solve the above mathematical programmings. Meanwhile, they have been implemented and integrated into the above modern solvers. Amongst kinds of mathematical programming, LP is the foundation. In other words, much performance enhancement of solver can be gained from the research of LP. Thus, this work aims to accelerate LP solving.

Recently, machine learning (ML) techniques have been used to improve mathematical programming solvers (NeurIPS 2021 Competition 2021) on a specific problem distribution. Because in real-life scenarios, a practitioner repeatedly solves problem instances from a specific distribution, with redundant patterns and similar characteristics. For example, managing a large-scale production planning among many factories requires solving very similar optimization problems on a daily basis, with relatively fixed manufacturing capabilities and transportation networks while only the demand of customers changes over time. This change in 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. A series of machine learning-based approaches have been proposed to improve the performance of the above solvers (Khalil et al. 2016; Gasse et al. 2019; Gupta et al. 2020; Nair et al. 2020).

Most of the above ML-enhanced approaches focus on using ML techniques to *replace some critical components within the solver*. Almost no previous work has thought of accelerating the solving by changing the mathematical formulation with the help of ML techniques. Because we unconsciously think that human experts are responsible for modeling and formulating real-life optimization problems. The expert-designed formulation is deemed the 'perfect' mathematical programming model and sent to the solver to get the optimal solution. Nevertheless, (Lodi and Tramontani 2013) discussed many perspectives of the *performance variability* in mixed-integer programming solver. Especially the formulation (such as the appearing order of variables of a given LP instance) is highly related to both the accuracy and solving speed of the solver, which leaves considerable space for improving the performance of solvers through reformulating the mathematical programming model. However, it is non-trivial to find the best appearing order of variables for a given LP

instance due to the permutation explosion.

To exploit the *performance variability*, we propose a novel approach to accelerate the LP solving from reformulation perspective via reinforcement learning. In our proposed approach, a graph convolutional neural network (GCNN) is firstly utilized to extract the patterns and characteristics of variables for a given LP. Then the pattern of variables is sent to a pointer network (PN) (Bello et al. 2016), from which we can get a new ordering of variables. In this way, we obtain a different but mathematically identical formulation of the original LP instance. The reformulation objective is to accelerate the solving process without decreasing the solving accuracy. The parameter of the above neural networks is trained via an end-to-end reinforcement learning method. Our contribution is three-fold:

- An easily overlooked phenomenon that the appearing order of variables in a given LP instance has a relatively significant impact on the solving performance of mathematical programming solver is revealed again in this paper, i.e., one of the performance variability of solvers.
- To exploit the above performance variability of solvers, we propose a reinforcement learning-based automatic reformulation approach. To the best of our knowledge, this is the first work to enhance the solver performance from reformulation perspective via reinforcement learning.
- Extensive experiments were performed on three representative and challenging datasets with three modern solvers, CLP, SCIP, and Gurobi. The results suggest that our method can effectively reduce the solving iteration number (20%↓) and the solving time (15%↓) on average, compared to directly solving the original LP instances.

2 Related work

Performance variability in mathematical programming solvers

A linear programming (LP) can be defined in the form of

$$\min_{\mathbf{x}} \mathbf{c}^T\mathbf{x}, \quad s.t. \quad \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0 \tag{1}$$

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; and $\mathbf{b} \in \mathbb{R}^m$ is the constraint right-hand-side vector; $\mathbf{x} \in \mathbb{R}^n$ is the variable vector; Note that formulation (1) is the standard form of linear programming. Any other form of linear programming can be transformed to the standard form (Maros 2002). Different from LP, mixed integer programming (MIP) has an extra integer constraint on a subset/all of variables. The most widely-adopted algorithms to solve linear programming include simplex method and interior point method (Maros 2002). These algorithms have been implemented in modern solvers. (Lodi and Tramontani 2013) discussed and analyzed the performance variability in the mathematical programming solvers. The performance of solvers is subject to some unexpected variability, for instance, the different computing platform, the permutation of rows and/or columns of a model and the addition of neutral changes to the solution process, etc. Lodi et. al. suggest that one source of the performance variability is rooted from the so-called *imperfect tie breaking*. Most of the decision made in solvers are based on *ordering candidates according to specifically-designed scores and selecting the best-scored candidates*. This is true for initial basis construction (Bixby 1992; Maros 2002) within the optimal face of LP solution process and also holds for cut separation and variable selection for MIP solution process. However, we are even not close to define what is the best scoring rules of decision making in solvers because it surely depends on what the scoring rule is used for, namely, initial basis construction, variable selection and cut selection, etc. Finally the consequence is that the algorithms just depend on the order in which the variables have been loaded or the floating-point computing arithmetics of the platform. An intuitive example for the performance variability of solvers is given in Section 3. We just exploit the performance variability to enhance the solving efficiency and robustness of linear programming solving, via resorting to machine learning techniques.

Graph representation of mathematical programmings

The relation between variables and constraints of mathematical programming can be represented by a bipartite graph, where a set of n nodes in the graph represents the n variables contained in the LP and the other set of m nodes correspond to the m constraints of the LP. The edge between a variable node and a constraint node represents the variable is present in the constraint. The number of edges indicates the number of non-zero coefficients of the constraint matrix $\mathbf{A}$. Other information such as the objective coefficients and constraint bounds, etc. can also be incorporated into the bipartite graph. In this way, the lossless representation of the LP can be sent as an input to graph neural networks. Many previous works adopt the representation method or related one to extract high-order embedding information of the original problems, such as Gasses et. al. and Nair et. al. done for mixed integer problem (Gasse et al. 2019; Nair et al. 2020), and Selsam et. al. done for satisfiability problem (Selsam et al. 2018).

Pointer network for combinatorial optimization

Combinatorial optimization problem such as Traveling Salesman Problem (TSP), Convex Hull problem, Set Cover problem, etc. play a fundamental role in the development of computer science, which have many applications in manufacturing, planning, genetic engineering, etc. Many kinds of algorithms have been proposed to solve above combinatorial optimization problem, including dynamic programming (Sumita et al. 2017; Chauhan, Gupta, and Pathak 2012), cutting plane (Applegate et al. 2003), local search (Zhang and Looks 2005) and neural network-based search method (Vinyals, Fortunato, and Jaitly 2015; Bello et al. 2016). In recent years, the application of neural networks on combinatorial optimization problem has drawn much more attention than other methods (Vinyals, Fortunato, and Jaitly 2015; Bello et al. 2016), especially after the proposal of Pointer Network (PN). The pointer network is a sequence-to-sequence model (Vinyals, Fortunato, and Jaitly 2015) originating from the field of natural language processing. It can learn the conditional probability of an output sequence of elements that are discrete

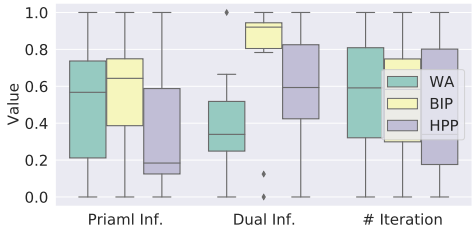


Figure 1: Three distinct LP instances (WA, BIP, and HPP) are selected to perform the preliminary experiment. For each original LP instance, we randomly changed the appearing order of variables to get many reformulated but mathematical identical LP instances. Next, we called Gurobi to solve the above LP instances. We recorded the primal infeasibility ('Primal Inf.'), dual infeasibility ('Dual Inf.') of the first iteration of Gurobi solving process, and the total iteration number ('# Iteration') to solve the instances. The significant variance of metrics shows that solver performance is quite sensitive to the different formulations for a given LP instance.

symbols corresponding to positions in an input sequence, which dedicates to dealing with variable size of output dictionary. On the TSP, (Vinyals, Fortunato, and Jaitly 2015) trained above neural network in a supervised manner to predict the sequence of visited cities. (Bello et al. 2016) trained the network with reinforcement learning method, using the negative tour length as the reward signal.

3 Preliminary Experiment and Motivation

Before the introduction of our solution, preliminary experiments were performed to reveal an easily overlooked phenomenon that the appearing order of variables in a given LP instance indeed affects the solver's performance. To this end, we selected three distinct LP instances (BIP, WA, and HPP) described statistically in Section 5. Three metrics, including initial primal infeasibility, initial dual infeasibility, and total iteration number, are recorded to show the effect of appearing order of variables on solving performance. Next, we randomly changed the appearing order of variables in a given LP instance to obtain many reformulated but mathematically identical LP instances. Then we called Gurobi solver to solve them. After normalizing above recorded [1] metrics into the range of $[0, 1]$, we visualize the normalized metrics in Figure 1. Observed from the large variance of the recorded metrics in Figure 1, we can infer that taking into input the same LP instances, the solver performance is quite sensitive to the different formulations (i.e., the different appearing order of variables in this paper) even with the one of state-of-the-art commercial solver Gurobi.

Therefore, it leaves us a relatively large room to improve the solver performance using machine learning techniques from a reformulation perspective. To that end, we have three incident challenges to tackle: 1) **How to appropriately represent an LP instance** in a low-dimension space without loss of relationships between variables, constraints, and objective of the LP instance; 2) **What machine learning model is suitable** to infer the best formulation of a given LP instance and 3) **How to efficiently train above inferring model** (if any) considering the permutation explosion of different formulations. Keeping in mind the challenges, we propose our reinforcement learning-based automatic reformulation method.

[1] Raw recorded metrics can be seen in Table 6 of Appendix.

4 Proposed Solution

Overview

In this section, our reformulation method is presented. We firstly introduce how we represent a given LP model and send it as input into a graph neural network. Then the embedding output by the graph neural network is aggregated with a given group of variable clusters and passed into a pointer network to get a new appearing order of variables. The appearing order is utilized to reformulate the original LP model to accelerate the solving process of the corresponding solver. The entire process is summarized in Figure 2.

Representation

We adopt the same method as done in (Gasse et al. 2019) to represent a given linear programming as a bipartite graph $(\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$. Specifically, in the bipartite graph, $\mathbf{C} \in \mathbb{R}^{m \times c}$ corresponds to the features of the constraints in the LP; $\mathbf{V} \in \mathbb{R}^{n \times d}$ denotes the features of the variables in the LP; and an edge $e_{ij} \in \mathbf{E}$ between a constraint node i and a variable node j if the corresponding coefficient $\mathbf{A}_{i,j} \neq 0$. For simplicity, we just attach the value of $\mathbf{A}_{i,j}$ to the corresponding edge e_{ij}. Readers can refer to the used features in Appendix.

Next, the bipartite graph representation of LP is sent as input into a two-interleaved graph convolutional neural network (GCNN) (Gasse et al. 2019). In detail, the graph convolution is broken into two successive passes, one from the variable side to the constraint side, and one from the constraint side to the variable side, which can be formulated as follows:

$$\mathbf{c}_i^{(k+1)} \leftarrow \mathbf{f}_{\mathbf{C}} \left(\mathbf{c}_i^{(k)}, \sum_{j}^{(i,j)\in\mathbf{E}} \mathbf{g}_{\mathbf{C}}(\mathbf{c}_i^{(k)}, \mathbf{v}_j^{(k)}, e_{ij}) \right), \quad (2)$$

$$\mathbf{v}_j^{(k+1)} \leftarrow \mathbf{f}_{\mathbf{V}} \left(\mathbf{v}_j^{(k)}, \sum_{i}^{(i,j)\in\mathbf{E}} \mathbf{g}_{\mathbf{V}}(\mathbf{c}_i^{(k)}, \mathbf{v}_j^{(k)}, e_{ij}) \right) \quad (3)$$

where $\mathbf{f}_{\mathbf{C}}, \mathbf{g}_{\mathbf{C}}, \mathbf{f}_{\mathbf{V}}$ and $\mathbf{g}_{\mathbf{V}}$ are 2-layer perceptrons with prenorm layer. We adopt the ReLU as the activation function. And k represents the number of times that we perform the convolution. The parameters of GCNN are denoted by θ_G.

Aggregation

The embedding of variables is obtained using the GCNN. However, we further perform an aggregation operation over the embeddings of variables rather than directly sending them to the pointer network. There are several reasons why we need to perform aggregation. First, the learning capability of the pointer network is limited. According to the evaluation report of previous work (Vinyals, Fortunato, and Jaitly 2015; Bello et al. 2016), the pointer network can achieve closely optimal results with up to 100 nodes in their experimental setting. It performs significantly worse when the number of nodes exceeds 1000. Second, considering all possible permutations of variables of a given LP is intractable. The number of

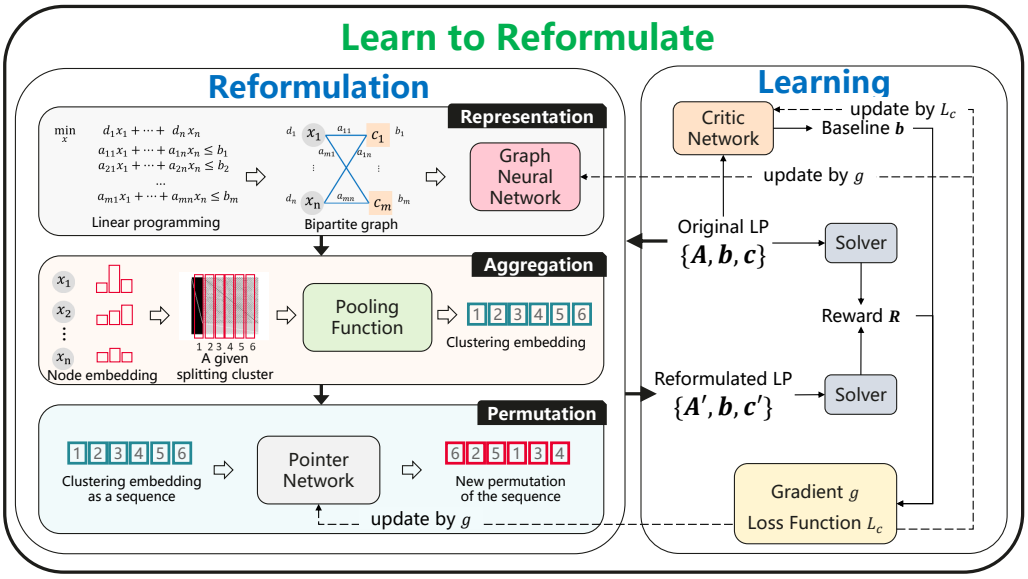


Figure 2: Overview of our proposed reinforcement learning-based automatic reformulation method, which is split into two parts, i.e., the reformulation and learning parts, respectively. The reformulation part can be summarized as three steps: a) the inputting LP instance is represented by a bipartite graph, and then the embedding of variables of the LP instance is obtained via a graph neural network (GNN); b) the embedding of variables will be further aggregated with a given group of variable clusters and c) taking as input the previous embeddings, a pointer network (PN) is used to output a new permutation of variables, i.e., the reformulation of the original LP instance. The learning part interacts with the reformulation part to update the parameters of GNN and PN, in the fashion of reinforcement learning.

LP variables that come from a practical scenario can easily exceed 100. Third, many LPs have their special structure, which can be exploited to split the variables into several clusters in advance. Many optimization methods such as Benders decomposition (Mavrotas and Makryvelios 2021; Gharaei, Karimi, and Hoseini Shekarabi 2020) have exploited the structure of the LP model to accelerate the solving process. Considering all the above, we perform the aggregation using the following steps:

Splitting. For a given LP as shown in (1), the variables $x_i (i = 1, ..., n)$ are split up into M disjoint clusters $C_j = \{x_{j_1}, x_{j_2}, ..., x_{j_{|C_j|}}\} (j = 1, ..., M)$. Note that variables within one cluster are subject to the order of variables in the original LP. The clustering method is not restricted. It could be specified by human experts or using the hyper-graph decomposition method (Manieri, Falsone, and Prandini 2021).

Pooling function. With above splitting clusters $C_j (j = 1, ..., M)$ and variable embedding $\mathbf{v}_i (i = 1, ..., n)$, we perform the aggregation for each cluster via:

$$\Sigma_j = \mathcal{P}(\{\mathbf{v}_i | x_i \in C_j\}) \tag{4}$$

where $\mathcal{P}$ is a pooling function that could be maximum, minimum, average, or other appropriate functions. We still do not restrict the kind of pooling function here. $\Sigma_j \in \mathbb{R}^d$ can be understood as the embedding of the splitting cluster.

Permutation

Given an LP l, we reformulate l by reordering its variables. More specifically, given a sequence of splitting clusters $\{C_j\}_{j=1}^M$ of variables of l, we would like to find a new permutation π of these clusters to reformulate l. The reformulation is achieved by that 1) between splitting clusters, the variables will be reordered with its associated cluster according to the permutation π; and 2) within each cluster, the order of variables remains the same with the original LP. In this way, the coefficients matrix $\mathbf{A}$ and objective coefficients vector $\mathbf{c}$ will correspondingly be altered. We hope the reformulation of the original LP can improve the solving performance of the solver over the LP. The solving performance can be the solving time, iteration number, primal/dual solution violation (Maros 2002), etc., which depends on the preference of performance optimization. We define the improvement R of solving performance gained from reformulation as:

$$R(\pi|l) = 1 - \frac{\mathcal{S}_{\mathcal{M}}(l|\pi)}{\mathcal{S}_{\mathcal{M}}(l)} \tag{5}$$

where $\mathcal{S}_{\mathcal{M}}(l)$ denotes that w.r.t. a solving performance metric $\mathcal{M}$, calling a solver to solve l; and $\mathcal{S}_{\mathcal{M}}(lp|\pi)$ refers to that w.r.t. the same solving metric $\mathcal{M}$, calling the same solver to solve l which is reformulated by the permutation π. Using Eq.(5), we can measure how the reformulation of l can improve the solving performance $\mathcal{M}$ of the solver, compared to directly solving the original LP l.

We aim to learn a probability distribution $p(\pi|\{C_j\}_{j=1}^M)$ that given a sequence of splitting clusters $\{C_j\}_{j=1}^M$ of LP l and associated embeddings $\{\Sigma_j\}_{j=1}^M$, can assign higher probabilities to "better" permutations and lower probabilities to "worse" ones. The "better" and "worse" are measured using Eq.(5). Similar to (Vinyals, Fortunato, and Jaitly 2015; Bello et al. 2016), the probability distribution $p(\pi|\{C_j\}_{j=1}^M)$ utilizes the chain rule to factorize the probability of a permu-

tation as:

$$p(\pi|\{C_j\}_{j=1}^M) = \prod_{j=1}^{M} p(\pi(j)|\pi(<j), \{C_j\}_{j=1}^M) \quad (6)$$

We parameterize $p(\pi|\{C_j\}_{j=1}^M)$ by a pointer network whose parameter is denoted by θ_P.

Training method

In our proposed method, there are two main groups of parameters, θ_G, and θ_P, to learn. Theoretically, the parameters can be learned using supervised learning (SL) as done in (Vinyals, Fortunato, and Jaitly 2015) or reinforcement learning (RL) as done in (Bello et al. 2016). However, we adopt the reinforcement learning method instead of the supervised learning method. First, getting high-quality labeled data (getting improvement R gained from reformulation in our context) is expensive, especially when the size of LP is large. Because we need to call a solver to solve two LP instances (i.e., original LP and its reformulated LP) each time we calculate the improvement R. Besides, RL is deemed as an effective way to generate better-supervised signals, which could help find a more competitive solution than purely supervised learning. Thus we propose to use model-free policy-based RL to learn θ_G and θ_P. The training objective is to maximize the expected improvement R over a given LP l, which is formally defined as:

$$J(\theta_G, \theta_P|l) = \mathbb{E}_{\pi\sim p_{\theta_G,\theta_P}(.|l)}[R(\pi|l)] \quad (7)$$

In our training phase, the LPs are *i.i.d.*. In other words, they usually originate from the same practical scenario such as production planning, bin packing, etc. We denote the distribution (or scenario) by $\mathcal{S}$. Thus the total training objective is defined as:

$$J(\theta_G, \theta_P) = \mathbb{E}_{l\sim\mathcal{S}}[J(\theta_G, \theta_P|l)] \quad (8)$$

We utilize stochastic gradient ascent method (Sebbouh, Cuturi, and Peyré 2022) to optimize Eq.(8). According to the REINFORCE algorithm (Li 2017), the gradient of Eq.(8) is given as:

$$\begin{aligned}\bigtriangledown_{\theta_G,\theta_P} J(\theta_G, \theta_P|l) = \mathbb{E}_{\pi\sim p_{\theta_G,\theta_P}(.|l)}[(R(\pi|l) - b(l)) \\ \bigtriangledown_{\theta_G,\theta_P} \log p_{\theta_G,\theta_P}(\pi|l)]\end{aligned} \quad (9)$$

where $b(.)$ is a baseline function independent of π and estimates the expected improvement R to reduce the learning variances. To enhance the estimated accuracy of baseline function, we additionally introduce a critic network parameterized by θ_c, which is trained with the stochastic gradient descent method over a mean squared error (MSE) between the actual improvement R and its prediction $b_{\theta_c}(l)$. The MSE loss is defined as:

$$\mathcal{L}(\theta_c) = \mathbb{E}_{l\sim\mathcal{S},\pi\sim p_{\theta_G,\theta_P}(.|l)}[b_{\theta_c}(l) - R(\pi|l)]^2 \quad (10)$$

Note that all mentioned-above gradients are approximated in our implementation using Monte Carlo sampling. We give a snippet of pseudo codes in Algorithm 1.

Algorithm 1: Training procedure of the proposed method

Inputs $\mathcal{S}$: set of linear programming problems; T: total number of episodes; B: batch size

Initializes θ_G: parameters for graph convolutional neural network; θ_P: parameters for pointer network; θ_c: parameters for critic network

for steps $t = 1$ to T **do**
  $l_i \sim \mathcal{S}$ for $i \in \{1, ..., B\}$
  $\pi_i \sim p_{\theta_G,\theta_P}(.|l_i)$ for $i \in \{1, ..., B\}$
  Calculate R_i using Eq.(5) for $i \in \{1, ..., B\}$
  $b_i = b_{\theta_c}(l_i)$ for $i \in \{1, ..., B\}$
  $g \leftarrow \frac{1}{B}\sum_{i=1}^{B}(R_i - b_i) \bigtriangledown_{\theta_G,\theta_P} \log p_{\theta_G,\theta_P}(\pi_i|l_i)$
  $\mathcal{L}_c \leftarrow \frac{1}{B}\sum_{i=1}^{B}(b_i - R_i)^2$
  Perform a gradient ascent step to update θ_G and θ_P using g respectively
  Perform a gradient descent step to update θ_c using $\bigtriangledown_{\theta_c}\mathcal{L}_c$
end for
return θ_G, θ_P and θ_c

5 Experimental Evaluation

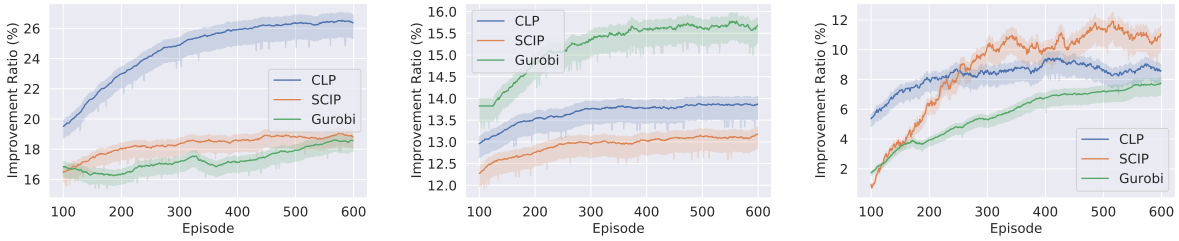
Implementation detail

Solvers and related interfaces. In our implementation, we adopt three well-established solvers (i.e., CLP, SCIP and Gurobi) as testbeds, with which the proposed reformulation method interacts. To facilitate the interaction between neural networks and solvers, we developed interfaces based on CyLP (COIN-OR Foundation 2021b), PySCIPOpt (Maher et al. 2016) and Gurobi Python API. The functions of these interfaces can be summarized as 1) taking as input a permutation outputted from neural networks; 2) reformulating an LP instance according to the given permutation; 3) solving the reformulated LP instance with a given solver, and 4) returning the solving metric of interest back to neural networks. Note that we set the **hyperparameters of solvers as their default values**, which are summarized in Table 5 (see Appendix).

Neural networks and aggregation. Concerning the GCNN and PN involved in the proposed method, we first refer to the design and implementation of (Gasse et al. 2019; Vinyals, Fortunato, and Jaitly 2015) respectively and then re-implemented them using PyTorch (Paszke et al. 2019). The corresponding **hyperparameters of above neural networks are also given in Table 5** (see Appendix). With respect to the aggregation, for a given LP, we split its variables into equal-size clusters according to their appearing order in the original LP. Besides, we adopt the average function as the pooling function.

Used computing resources. All experiments were conducted on a computing server, equipped with Intel(R) Xeon(R) Platinum 8180M CPU@2.50GHz, a V100 GPU card with 32GB graphic memory and 1TB main memory. For each training and each testing, they were performed five times with different random seeds. Each training with 500 episodes took 12.5 hours of CPU-GPU computing on average. The total amount of computing consumption is around 675 CPU-GPU hours.

Dataset All experiments were performed over three sets of Mixed Integer Linear Programming (MILP) problems. All datasets are scenario specific. In other words, they contain problem instances from only a single scenario. Two of them, Balanced Item Placement (BIP) and Workload Apportionment (WA), are from NeurIPS 2021 ML4CO compe-



(a) BIP (b) WA (c) HPP

Figure 3: Learning convergence and improvement of the iteration number over three datasets' training (seen) instances. Several findings can be pointed out: 1) the networks' parameters can converge over the three datasets; 2) with each testing solver, our method is effective in reducing the solving iteration number; 3) on the LP instances from WA and HPP, our method performs slightly worse than those from BIP, which demonstrates that it is relatively harder to learn neural network parameter over the large-scale and complex LP instances.

Table 1: Statistical description of used dataset

Dataset	m	n	NNZ
BIP	195.00 ± 00.00	1083.00 ± 00.00	7440.00 ± 00.00
WA	$6.43e04 \pm 54.51$	$6.1e04 \pm 00.00$	$3.62e05 \pm 6007.41$
HPP	$1.25e06 \pm 6.93e04$	$2.66e06 \pm 1.83e05$	$6.64e06 \pm 4.29e05$

tition (NeurIPS 2021 Competition 2021). The third one is obtained from a real-life production planning scenario called HPP. The detail about these datasets is given in the Appendix, where we describe the physical meaning of the problem, such as optimization objective and constraints. Note that we *relax* the integer constraint of variables in the above MILPs to get the corresponding LP instances. Concerning the dataset of BIP and WA, there are in total $10,000$ LP instances, respectively, which are different in the value of coefficients, the size of constraints, and variables. Moreover, for the dataset of HPP, there are in total $1,000$ LP instances. The statistical description of the above datasets is summarized in Table 1, where m, n, and NNZ represent the number of constraints, variables, and non-zero coefficients, respectively. For each dataset, 80% and 20% of instances are randomly selected into the training and testing set, respectively. The instance selection is subject to the uniform distribution. It should admit that our method can only generalize over *i.i.d.* LP instances. Thus, we train the neural networks in the proposed method separately for each dataset. In other words, we have three sets of $(\theta_G, \theta_P, \theta_c)$ to train. Enhancing the generalization of the proposed method by meta learning (Hospedales et al. 2020) is left to future work.

Metric of interest As defined in Eq.(5), we need to specify the solving performance metric $\mathcal{M}$ when we measure the improvement gained from reformulation. In practice, people usually care about the solving time or iteration number of the search process when the solution quality meets a given standard (for example, the maximum of primal/dual infeasible max_inf of a solution is less than a given primal/dual tolerance). Nevertheless, the solving time is very sensitive to the run-time state of the computing server where many other background tasks simultaneously run. The training process will be unstable if we adopt solving time as our metric. Luckily, the solving time is empirically proportional to the solving iteration, which is not affected by the run-time state of the computing server. Thus we finally adopt the iteration number as the solving performance metric $\mathcal{M}$. Besides, we also specific requirement for solution quality, that is $max\_inf \leq 10^{-6}$.

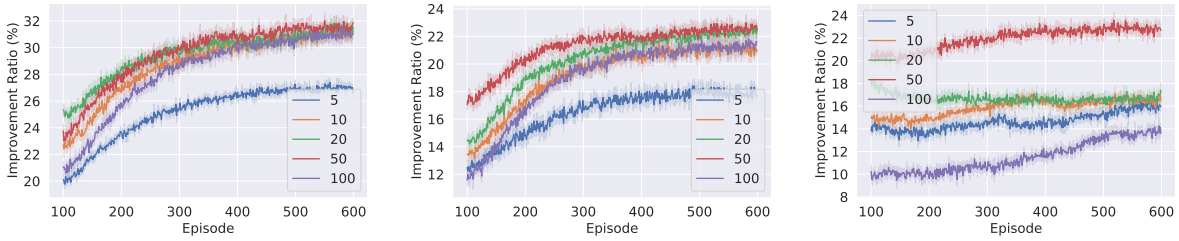
Table 2: Improvement of the iteration number (the lower, the better) over testing instances of three datasets.

Dataset	Solver	Avg. # Iteration		Improv. (%)
		Original	Reformulated	
BIP	CLP	956.32	705.10	26.27
	SCIP	673.58	545.33	19.04
	Gurobi	525.46	428.30	18.49
WA	CLP	6943.37	5985.18	13.8
	SCIP	11354.34	9874.87	13.03
	Gurobi	17962.85	15110.35	15.88
HPP	CLP	98352.27	90277.55	8.21
	SCIP	94124.48	82933.08	11.89
	Gurobi	23526.64	21656.27	7.95

Evaluation result

Improvement on solving iteration number We measure how the proposed reformulation method reduces the solving iteration number compared to directly solving the original LP. Specifically, we first call the solvers to solve original LP instances from the above three datasets and record the iteration number (here refers to the iteration number of the dual simplex method). Then we reformulate these LP instances using the learned neural networks. The same solver is called again to solve the reformulated LP instances, and the iteration number is recorded subsequently. We compare the iteration number of solving reformulated LP instances against that of solving original ones.

The learning convergence and comparison results are presented in Figure 3 and Table 2. In Figure 3, the vertical axis, improvement ratio, denotes the improvement of solving performance gained from reformulation (i.e., the reduction ratio of iteration number of reformulation to that of original one). Several findings can be pointed out: 1) the parameter learning of neural networks involved in our method can converge over the three datasets; 2) with each testing solver, our reformulation method is effective in reducing the solving iteration number; 3) on the LP instances from WA and HPP, our method performs slightly worse than those from BIP, which demonstrates that it is relatively harder to learn neural network parameter over the large-scale and complex LP instances from WA and HPP. Besides, in Table 2, it can be demonstrated that our method can be generalized to testing



(a) CLP over BIP (b) SCIP over BIP (c) Gurobi over BIP

Figure 4: How different number of clustering block impacts our method's performance on BIP. Several findings can be pointed out: 1) it can be noted that increasing the number of clustering block within a specific range will lead to an improvement in the performance of our learning-based method; 2) it can also suffer a dramatic performance degradation if the number of clustering blocks is too large. Similar conclusions are drawn on the other two datasets (see Figure 6 in Appendix).

Table 3: Improvement of solving time (the lower, the better) by our proposed method over all instances of three datasets

Dataset	Seen/Unseen	Solver	Avg. Solving Time (s)		Avg. Reordering Time (s)	Improv. (%)
			Original	Reformulated		
BIP	Seen	CLP	0.03862	0.02851	0.00011	26.18
		SCIP	0.04625	0.03453	0.00012	25.35
		Gurobi	0.01086	0.00849	0.00012	21.78
	Unseen	CLP	0.01792	0.01355	0.00019	24.39
		SCIP	0.04636	0.03472	0.00017	25.11
		Gurobi	0.01144	0.00939	0.00018	17.86
WA	Seen	CLP	6.74099	5.76085	0.00695	14.54
		SCIP	3.96057	3.39302	0.00811	14.33
		Gurobi	5.21913	4.23533	0.00723	18.85
	Unseen	CLP	7.17389	6.17385	0.00689	13.94
		SCIP	4.04960	3.29840	0.00709	18.55
		Gurobi	4.31686	3.59897	0.00685	16.63
HPP	Seen	CLP	71.0665	63.7396	0.01441	10.31
		SCIP	40.4433	34.2676	0.01719	15.27
		Gurobi	43.2238	37.7647	0.01996	12.63
	Unseen	CLP	70.9742	65.9208	0.01334	7.12
		SCIP	40.7626	34.9947	0.01755	14.15
		Gurobi	43.9299	40.0245	0.01454	8.89

(unseen) data since it still performs well over the testing data.

Improvement of solving time We continue to measure how our proposed method reduces the solving time. The same procedure as described in Section 5 is adopted in this experiment. Unlike Section 5, we here compare the solving time between original LP instances and its reformulation obtained from our method instead of the solving iteration number. The results are recorded in Table 3. Observing the results, several claims can be made: 1) the proposed method can indeed reduce the solving time of given LP instances by reformulating them, which inferably benefits from the reduction of solving iteration number; 2) our method performs slightly worse over the complex and large-scale LP instances but still can reduce at least 10.31% the solving time over the complex LP instances from WA and HPP; 3) our method does not depend on the type of solver since the performance improvements are achieved on all three solvers in the experiments; 4) the learned neural networks still can be generalized to testing (unseen) data, and 5) the reordering process including the inference of neural networks and model reformulation is swift compared with the solving process. Besides, to figure out what the neural networks have learned, visualization for the reformulation process of our method is given (see visual analysis part in Appendix).

Impact of different cluster size Further, we investigate how cluster size impacts performance. We demonstrate the performance comparison with the different number of clustering blocks (1, 5, 10, 20, 50, and 100). When there is only one clustering block, no reformulation is made. Thus the improvement ratio is always zero in the training process. The same procedure as described in Section 5 is adopted in this experiment. Results [2] are presented in Figure 4. Several findings can be pointed out: 1) it can be noted that increasing the number of clustering blocks within a certain range will lead to an improvement in the performance of our learning-based method; 2) it can also suffer a dramatic performance degradation if the number of clustering blocks is too large. Moreover, it is in accordance with our common intuition. Specifically, the proper number of clustering blocks of BIP is referred to in the range of 50~100; the one of WA is referred over 100; the one of HPP is referred to in the range of 1~5. These results can help to determine the proper clustering size when solving LP problems from the same distribution.

6 Conclusion

In this paper, we propose a reinforcement learning-based reformulation method to accelerate the linear programming solving. It makes use of the graph neural network and pointer network together to find the better reformulation for linear programmings that come from a specific distribution. In the detailed computational studies, we demonstrated that the formulation derived from our proposed approach effectively reduced the solving iteration and solving time, compared to original formulation of LP instances, which is independent of solvers. To the best of our knowledge, this is the first work that exploits the performance variability of modern solvers via machine learning techniques to gain performance.

The idea of using machine learning techniques to exploit or to reduce the performance variability of mathematical programming solvers can be extended in many directions. First, one can further learn the modeling experience or gain the modeling tricks from the reformulation derived from the neural networks. Besides, one can use the proposed method to decide the better ordering rules in various decision-making components in solver such as pricing, variable selection, cut separation, etc. We believe that this work can inspire the future research to better exploit the performance variability of solvers to improve the solvers.

[2] Experimental results of WA and HPP are placed in Figure 6 in Appendix due to the space limitation.

References

Andersen, E. D.; Roos, C.; and Terlaky, T. 2003. On implementing a primal-dual interior-point method for conic quadratic optimization. *Mathematical Programming*, 95(2): 249–277.

Applegate, D.; Bixby, R.; Chvátal, V.; and Cook, W. 2003. Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Mathematical programming*, 97(1): 91–153.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

Bixby, R. E. 1992. Implementing the simplex method: The initial basis. *ORSA Journal on Computing*, 4(3): 267–284.

Chauhan, C.; Gupta, R.; and Pathak, K. 2012. Survey of methods of solving tsp along with its implementation using dynamic programming approach. *International journal of computer applications*, 52(4).

COIN-OR Foundation. 2021a. COIN-OR Linear Programming, https://github.com/coin-or/Clp,.

COIN-OR Foundation. 2021b. CyLP, https://github.com/coin-or/CyLP,.

Dantzig, G. B. 1987. Origins of the Simplex Method.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint arXiv:1906.01629*.

Ge, D.; Wang, C.; Xiong, Z.; and Ye, Y. 2021. From an Interior Point to a Corner Point: Smart Crossover.

Gharaei, A.; Karimi, M.; and Hoseini Shekarabi, S. A. 2020. Joint economic lot-sizing in multi-product multi-level integrated supply chains: Generalized benders decomposition. *International Journal of Systems Science: Operations & Logistics*, 7(4): 309–325.

Gupta, P.; Gasse, M.; Khalil, E. B.; Kumar, M. P.; Lodi, A.; and Bengio, Y. 2020. Hybrid models for learning to branch. *arXiv preprint arXiv:2006.15212*.

Gurobi. 2021. Gurobi Solver, https://www.gurobi.com/,.

Hospedales, T.; Antoniou, A.; Micaelli, P.; and Storkey, A. 2020. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*.

IBM. 2021. CPLEX, https://www.ibm.com/hk-en/analytics/cplex-optimizer,.

Khalil, E.; Le Bodic, P.; Song, L.; Nemhauser, G.; and Dilkina, B. 2016. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

Li, X.; Han, X.; Zhou, Z.; Yuan, M.; Zeng, J.; and Wang, J. 2021. Grassland: A Rapid Algebraic Modeling System for Million-variable Optimization. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 3925–3934.

Li, Y. 2017. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*.

Lodi, A.; and Tramontani, A. 2013. Performance variability in mixed-integer programming. In *Theory driven by influential applications*, 1–12. INFORMS.

Maher, S.; Miltenberger, M.; Pedroso, J. P.; Rehfeldt, D.; Schwarz, R.; and Serrano, F. 2016. PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite. In *Mathematical Software – ICMS 2016*, 301–307. Springer International Publishing.

Manieri, L.; Falsone, A.; and Prandini, M. 2021. Hyper-graph partitioning for a multi-agent reformulation of large-scale MILPs. *IEEE Control Systems Letters*, 6: 1346–1351.

Maros, I. 2002. *Computational techniques of the simplex method*, volume 61. Springer Science & Business Media.

Mavrotas, G.; and Makryvelios, E. 2021. Combining multiple criteria analysis, mathematical programming and Monte Carlo simulation to tackle uncertainty in Research and Development project portfolio selection: A case study from Greece. *European Journal of Operational Research*, 291(2): 794–806.

Nair, V.; Bartunov, S.; Gimeno, F.; von Glehn, I.; Lichocki, P.; Lobov, I.; O'Donoghue, B.; Sonnerat, N.; Tjandraatmadja, C.; Wang, P.; et al. 2020. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*.

NeurIPS 2021 Competition. 2021. Machine Learning for Combinatorial Optimization, https://www.ecole.ai/2021/ml4co-competition/,.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32: 8026–8037.

Pochet, Y.; and Wolsey, L. A. 2006. *Production planning by mixed integer programming*. Springer Science & Business Media.

Sebbouh, O.; Cuturi, M.; and Peyré, G. 2022. Randomized Stochastic Gradient Descent Ascent. In *International Conference on Artificial Intelligence and Statistics*, 2941–2969. PMLR.

Selsam, D.; Lamm, M.; Bünz, B.; Liang, P.; de Moura, L.; and Dill, D. L. 2018. Learning a SAT solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*.

Sumita, H.; Yonebayashi, Y.; Kakimura, N.; and Kawarabayashi, K.-i. 2017. An Improved Approximation Algorithm for the Subpath Planning Problem and Its Generalization. In *IJCAI*, volume 2017, 4412–4418.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. *arXiv preprint arXiv:1506.03134*.

Wolsey, L. A. 2007. Mixed integer programming. *Wiley Encyclopedia of Computer Science and Engineering*, 1–10.

Zhang, W.; and Looks, M. 2005. A novel local search algorithm for the traveling salesman problem that exploits backbones. In *IJCAI*, volume 5, 343–384. Citeseer.

ZIB. 2021. SCIP Solver, https://www.scipopt.org/,.

A Appendix

Feature used in constructing bipartite graph

Table 4: Used features of constraints, variables and edges in the bipartite graph

Tensor	Feature	Meanings
C	rhs	the right-hand side coefficients of LP, i.e. **b**, normalized with constraint coefficients
	ub_cons	upper bound of constraint, normalized with all constraints upper bound
	lb_cons	lower bound of constraint, normalized with all constraints lower bound
V	obj	the objective coefficients of variables, i.e. **c**
	lb_var	upper bound of variable, normalized with all variables lower bound
	ub_var	lower bound of variable, normalized with all variables upper bound
E	coef	the constraint coefficients of variables, i.e. **A** , normalized per constraint

Description of used datasets

Balanced Item Placement. This problem deals with spreading items (e.g., files or processes) across containers (e.g., disks or machines) and 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.

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.

Huawei Production Planning. The planning and scheduling optimization problems are solved in the Huawei production planning engine. The production planning problem is to plan daily production for hundreds of factories according to customers' daily and predicted demand. Besides, the problem is subject to material transportation and production capacity constraints. The problem's optimization objective is to minimize the production cost and lead time simultaneously. Readers can refer to (Li et al. 2021; Pochet and Wolsey 2006).

Hyper-parameters setting

Part of important hyperparameters involved in our method is listed in Table 5.

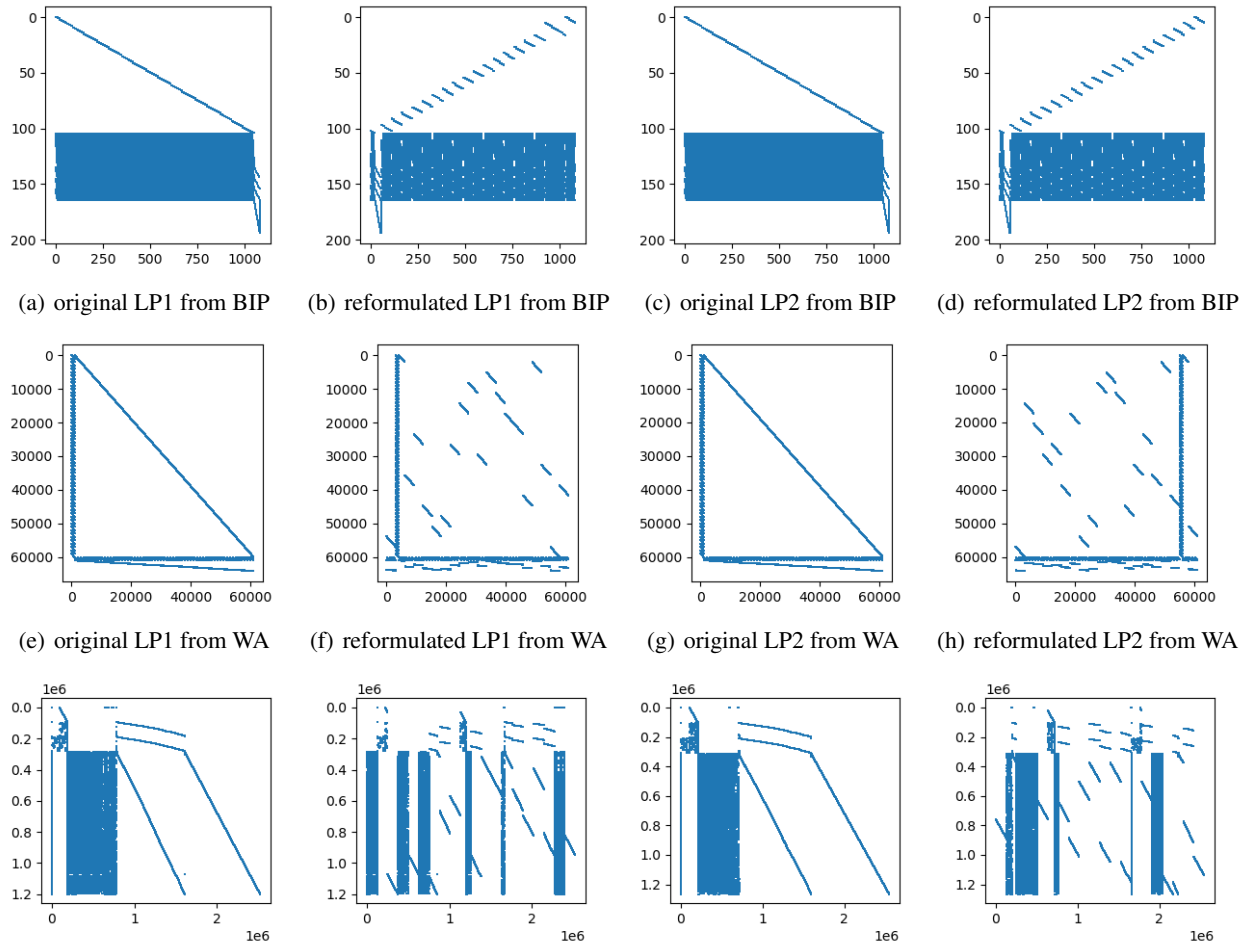
Table 5: Hyperparameters setting

Name	Used value
Optimizer	*ADAM*
# episode (T)	500
# splitting cluster	20
Batch size (B)	8
Train size	640
Validation size	320
Learning rate	10^{-4}
Decay ratio of learning rate	0.96
Gradient clip normalizer	l_2 Normalization
Dimension of input embedding in PN	128
Dimension of hidden layers in PN	128
Dimension of input embedding in GCNN	64
# of times that performs convolution	2
Presolve of CLP	On
Presolve of SCIP	On
Presolve of Gurobi	On
Solving method of CLP	Dual
Solving method of SCIP	Dual
Solving method of Gurobi	Dual

Visual analysis

In order to figure out what the neural network has learned, we give the visualization for the reformulation process of our method over the three datasets of LP instances. Specifically, we select two LP instances from each dataset to visualize. We reformulate them by the learned neural network of our proposed method. Then we visualize the coefficient matrix of original LP instances and reformulated ones, respectively, which are shown in Figure 5(a) to Figure 5(l).

Observing these figures, we can find that 1) our proposed reformulation method captures the characteristics of LP instances originating from different scenarios. Because the pattern of corresponding reformulated LP instances are significantly different across different datasets but are similar between LP instances within the same dataset; 2) The reformulation is relatively stable when the original LP instances are highly similar. All the original LP instances of BIP have the same number of constraints and variables, which is only different in the value of coefficients. Thus the pattern of the corresponding reformulated LP instances are almost the same (see Figure 5(a) to Figure 5(d)). However, the pattern of reformulated LP instances of WA and HPP is quite different (see Figure 5(e) to Figure 5(h) and Figure 5(i) to Figure 5(l)) because the corresponding original LP instances differ in not only the value of coefficients but also the number of constraints and variables.



(a) original LP1 from BIP (b) reformulated LP1 from BIP (c) original LP2 from BIP (d) reformulated LP2 from BIP

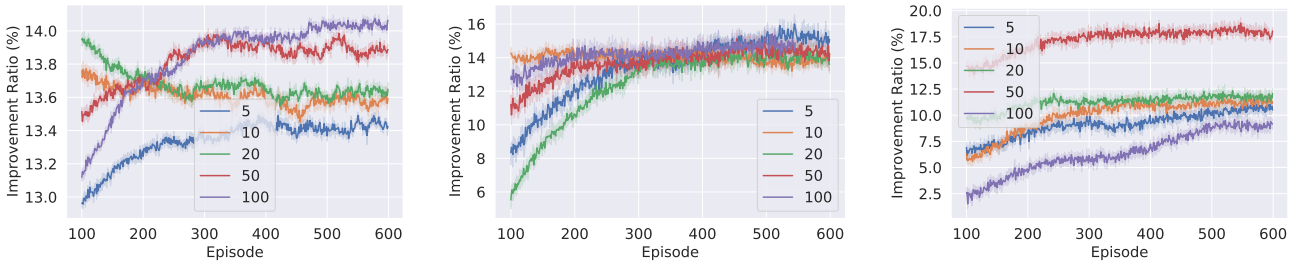
(e) original LP1 from WA (f) reformulated LP1 from WA (g) original LP2 from WA (h) reformulated LP2 from WA

(i) original LP1 from HPP (j) reformulated LP1 from HPP (k) original LP2 from HPP (l) reformulated LP2 from HPP

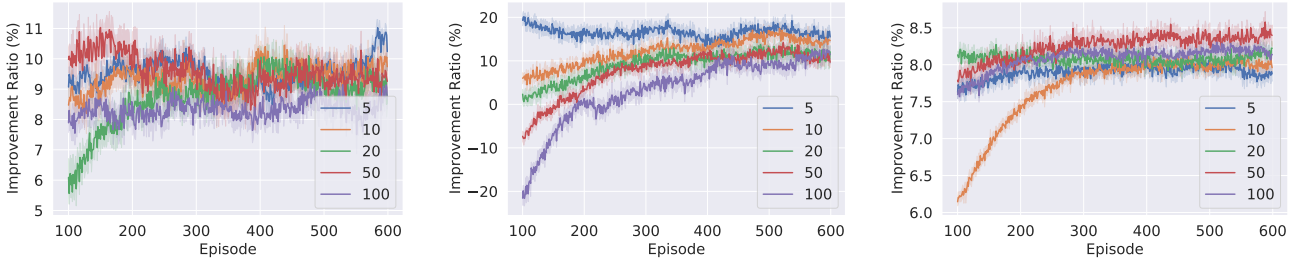
Figure 5: Visualization of reformulation process over BIP, WA and HPP datasets

Table 6: Statistics of preliminary experiments. 'Primal Inf.' and 'Dual Inf.' respectively indicates the primal infeasible value and dual infeasible value at the first iteration of Gurobi Solver (using default dual simplex method), i.e., the initial solution quality. And the '# Iteration' denotes the total iteration number that the solver used to solve a given instance.

Idx	HPP (m=146722, n=260636, nnz=668270)			WA (m= 64282, n=61000, nnz=359428)			BIP (m=195, n=1083, nnz=7440)		
	Primal Inf.	Dual Inf.	# Iteration	Primal Inf.	Dual Inf.	# Iteration	Primal Inf.	Dual Inf.	# Iteration
1	7.721295E+09	1.884180E+11	19408.00	311.22	1.9980E+09	14083.00	9.43	1.922493E+07	539.00
2	7.720309E+09	1.882416E+11	19242.00	295.33	1.9790E+09	17020.00	9.58	1.937180E+07	486.00
3	7.725508E+09	1.869514E+11	19208.00	269.29	1.9530E+09	16495.00	9.38	1.916960E+07	557.00
4	7.764919E+09	1.876644E+11	19408.00	262.35	2.0620E+09	15149.00	9.47	1.932945E+07	453.00
5	7.726977E+09	1.878485E+11	19175.00	299.10	1.9080E+09	16351.00	9.37	1.915617E+07	586.00
6	7.746786E+09	1.877944E+11	19266.00	281.19	1.8650E+09	14971.00	9.43	1.922493E+07	528.00
7	7.746866E+09	1.879217E+11	19274.00	308.00	1.9630E+09	15778.00	9.22	1.706866E+07	591.00
8	7.729257E+09	1.873669E+11	19391.00	269.86	2.1070E+09	16586.00	9.00	1.880115E+07	519.00
9	7.745738E+09	1.875424E+11	19203.00	296.18	2.0280E+09	14984.00	9.22	1.902808E+07	453.00
10	7.727762E+09	1.882426E+11	19240.00	284.81	2.2290E+09	15861.00	9.05	1.674231E+07	554.00



(a) CLP over WA (b) SCIP over WA (c) Gurobi over WA



(d) CLP over HPP (e) SCIP over HPP (f) Gurobi over HPP

Figure 6: How different number of clustering block impacts the our method's performance on WA and HPP