G2MILP: Learning to Generate Mixed-Integer Linear Programming Instances for MILP Solvers

Jie Wang ¹, Zijie Geng ², Xijun Li ¹, Jianye Hao ¹, Yongdong Zhang ¹, and Feng Wu ¹

¹Affiliation not available
²University of Science and Technology of China

December 7, 2023

Abstract

There have been significant efforts devoted to developing advanced mixed-integer linear programming (MILP) solvers, which are powerful tools for solving various real-world optimization problems. Despite the achievements, the limited availability of real-world instances often results in sub-optimal decisions and biased evaluations, which motivates a suite of MILP instance generation techniques. However, these approaches either rely on expert-designed formulations or struggle to capture the rich features of real-world instances. Moreover, the task of generating challenging MILP instances—which are valuable resources for evaluating solvers and motivating more efficient algorithms—remains underexplored. To tackle these problems, we propose G2MILP, which to the best of our knowledge is the first deep generative framework for MILP instances. Specifically, G2MILP represents MILP instances as bipartite graphs and employs a masked variational autoencoder to iteratively corrupt and replace parts of the original graphs to generate new ones. We then propose a hardness-oriented scheme, which iteratively augments the generator by learning from the hardest instances, to enhance G2MILP to construct challenging MILP instances. Experiments demonstrate that G2MILP can generate realistic MILP instances to effectively facilitate downstream tasks. Moreover, G2MILP can generate difficult instances initializing from given datasets, and the boost of hardness can be orders of magnitude.
G2MILP: Learning to Generate Mixed-Integer Linear Programming Instances for MILP Solvers

Jie Wang*, Senior Member, IEEE, Zijie Geng, Xijun Li, Jianye Hao, Member, IEEE, Yongdong Zhang, Senior Member, IEEE, and Feng Wu, Fellow, IEEE

Abstract—There have been significant efforts devoted to developing advanced mixed-integer linear programming (MILP) solvers, which are powerful tools for solving various real-world optimization problems. Despite the achievements, the limited availability of real-world instances often results in sub-optimal decisions and biased evaluations, which motivates a suite of MILP instance generation techniques. However, these approaches either rely on expert-designed formulations or struggle to capture the rich features of real-world instances. Moreover, the task of generating challenging MILP instances—which are valuable resources for evaluating solvers and motivating more efficient algorithms—remains underexplored. To tackle these problems, we propose G2MILP, which to the best of our knowledge is the first deep generative framework for MILP instances. Specifically, G2MILP represents MILP instances as bipartite graphs and employs a masked variational autoencoder to iteratively corrupt and replace parts of the original graphs to generate new ones. We then propose a hardness-oriented scheme, which iteratively augments the generator by learning from the hardest instances, to enhance G2MILP to construct challenging MILP instances. Experiments demonstrate that G2MILP can generate realistic MILP instances to effectively facilitate downstream tasks. Moreover, G2MILP can generate difficult instances initializing from given datasets, and the boost of hardness can be orders of magnitude.

Index Terms—Mixed-Integer Linear Programming, Instance Generation, Graph Generation, Variational Autoencoder.

1 INTRODUCTION

MIXED-INTEGER linear programming (MILP)—a powerful and versatile modeling technique for many real-world problems—lies at the core of combinatorial optimization (CO) research. It finds wide-ranging applications in various industrial optimization scenarios, such as scheduling [1], planning [2], and portfolio [3]. However, solving MILPs, which are known as NP-hard problems [4], presents significant challenges. Extensive efforts have been devoted to developing advanced MILP solvers, such as Gurobi [5], SCIP [6], and OptVerse [7]. In particular, machine learning (ML) techniques have emerged as a potent approach for solving MILPs or assisting the solving process [8], [9]. Notable successes include [10] for node selection, [11] for branching decision, and [12] for cut selection, etc.

The collection of MILP instances plays a fundamental role in developing advanced MILP solvers, primarily from three aspects [13]. First, for tasks such as hyperparameter configuration and ML model training, a substantial and diverse set of realistic and independent identically distributed (i.i.d.) instances is necessary. Second, the evaluation of solvers requires as many instances as possible to identify potential issues and weaknesses through white-box testing [5]. Finally, to inspire more efficient algorithms and promote research investigations, the research community keeps calling for challenging MILP instances for better benchmarking and competitions [13].

However, the limited availability of real-world instances, due to labor-intensive data collection and proprietary issues, remains a critical challenge and often leads to sub-optimal decisions and biased assessments [8], [13], [14]. This challenge motivates a suite of synthetic MILP instance generation techniques, which fall into two categories. Some methods rely heavily on expert-designed formulations for specific problems, such as Traveling Salesman Problems (TSPs) [15] or Set Covering problems [16]. However, these methods cannot cover real-world applications where domain-specific expertise or access to the combinatorial structures is limited. Other methods construct general MILP instances by sampling from an encoding space that controls a few specific statistics [17]. However, these methods often struggle to capture the rich features and the underlying combinatorial structures, resulting in an unsatisfactory alignment with real-world instances. Moreover, these works have scarcely investigated the ability of the generators to produce challenging MILP instances.

Developing deep learning (DL)-based generators is a promising way to address the challenge of limited data availability for the following reasons. First, these generators can actively learn from real-world instances and generate new ones without relying on expert-designed formulations. The generated instances can simulate realistic scenarios, cover a wide range of cases, significantly enrich datasets, and facilitate the development of MILP solvers at a relatively low cost. Second, learning-based generators have the capacity to capture instance features, enabling them to efficiently explore the space of MILPs to construct challenging instances. Finally, this approach shows promising technical prospects for un-

---

* J. Wang, Z. Geng, X. Li, Y. Zhang, F. Wu are with: a) CAS Key Laboratory of Technology in GIPAS, University of Science and Technology of China, Hefei 230027, China; b) Institute of Artificial Intelligence, Hefei Comprehensive National Science Center, Hefei 230091, China. E-mail: jiewang@ustc.edu.cn, {ustcgzj, lixijun}@mail.ustc.edu.cn, {zhyd73, fengwu}@ustc.edu.cn.

† X. Li and J. Hao are with Huawei Noah’s Ark Lab. E-mail: {xijun.li, haojiange}@huawei.com.

Manuscript received November, 2023. *Corresponding author.
In this paper, we propose G2MILP, which to the best of our knowledge is the first deep generative framework for MILP instances. We represent MILP instances as weighted bipartite graphs, where variables and constraints are vertices, and non-zero coefficients are edges. This graph representation enables us to use graph neural networks (GNNs) to effectively capture the essential features of MILP instances using graph neural networks (GNNs) [11], [19]. In this way, we recast the original task as a graph generation problem. To accommodate various application scenarios, we consider two task settings for utilizing G2MILP in MILP instance generation: realistic MILP instance generation and hard MILP instance generation (see Figure 1). We begin by focusing on realistic MILP instance generation, where our objective is to generate new MILP instances that closely resemble real-world instances in terms of their structures and computational hardness. Since generating the complex bipartite graphs from scratch can be computationally expensive and may destroy the intrinsic combinatorial structures of the problems [20], we propose a masked variational autoencoder (VAE) paradigm inspired by the masked autoencoder (MAE) [21] and the variational autoencoder (VAE) theories [22], [23], [24]. This paradigm iteratively corrupts and replaces parts of the original graphs using sampled latent vectors. To implement the complicated generation steps, we design a decoder consisting of four modules that work cooperatively to determine multiple components of new instances, encompassing both structure and numerical prediction tasks simultaneously. Subsequently, we work on hard MILP instance generation, where our goal is to construct challenging MILP instances. To achieve this, we propose a hardness-oriented iterative augmenting scheme. In each iteration, G2MILP generates a batch of new instances and stores the most difficult instances in a storage. We then fine-tune G2MILP using the instances in the storage as a training set, so that the model is specifically oriented towards generating challenging instances.

We design a suite of benchmarks to evaluate the quality of instances generated by G2MILP. Specifically, for realistic MILP instance generation, we demonstrate that G2MILP is the very first method capable of generating instances that closely resemble the training sets in terms of both structures and computational hardness. Moreover, we conduct a predictive downstream task to demonstrate the potential of generated instances in enhancing MILP solvers. The results show that using the generated instances to enrich the training sets reduces the prediction error by over 20% on several datasets. For hard MILP instance generation, the results show that G2MILP can generate very hard instances initializing from given datasets, and the boost of hardness can be orders of magnitude.

An earlier version of this paper has been published at NeurIPS 2023 [25]. This journal manuscript significantly extends the conference version by introducing a novel task setting, i.e., hard MILP instance generation (see Figure 1). This task is important as it provides valuable resources for the research community to evaluate solvers and thus potentially motivates more efficient algorithms [13]. Moreover, it demonstrates an important aspect of deep generative models’ capability in goal-directed generation. The incorporation of the task holds great promise to open up new avenues for the future research in the field of MILP. We enhance G2MILP by proposing a hardness-oriented iterative augmenting scheme to address this task (see Section 3.3). To the best of our knowledge, G2MILP is the first generation approach capable of learning to generate MILP instances with the explicit objective of constructing hard instances. Experiments on several datasets demonstrate its remarkable effectiveness (see Section 4.3).

2 RELATED WORK

2.1 Machine Learning for MILP

Machine learning (ML) techniques, due to its capability of capturing rich features from data, has shown impressive potential in addressing combinatorial optimization (CO) problems [26], [27], [28], especially MILP problems [8]. Some works apply ML models to directly solve MILPs [29], [30], [31]. Others attempt to incorporate ML models into heuristic components in modern solvers [10], [12], [32], [33]. Gasse et al. [11] proposed to represent MILP instances as bipartite graphs, and use graph neural networks (GNNs) to capture features for branching decisions. G2MILP can produce novel instances to enrich the datasets, which promises to enhance
the existing ML methods that require large amounts of independently identical distributional data.

2.2 MILP Instance Generation

Many previous works have made efforts to generate synthetic MILP instances for developing and testing solvers. Existing methods fall into two categories. The first category focuses on using mathematical formulations to generate instances for specific combinatorial optimization problems such as TSP [15], set covering [16], and mixed-integer knapsack [34]. The second category aims to generate general MILP instances. Bowly [17] proposed a framework to generate feasible and bounded MILP instances by sampling from an encoding space that controls a few specific statistics, e.g., density, node degrees, and coefficient mean. However, the aforementioned methods either rely heavily on expert-designed formulations or struggle to capture the rich features of real-world instances. G2MILP tackles these two issues simultaneously by employing deep learning techniques to actively generate instances that resemble real-world problems, and it provides a versatile solution to the data limitation challenge.

2.3 Deep Graph Generation

A plethora of literatures have investigated deep learning models for graph generation [35], including auto-regressive methods [36], variational autoencoders (VAEs) [23], and generative diffusion models [37]. These models have been widely used in various fields [38] such as molecule design [39], [40], [41] and social network generation [42], [43]. G2SAT [18], the first deep learning method for SAT instance generation, has received much research attention [20], [44]. Nevertheless, it is non-trivial to adapt G2SAT to MILP instance generation, as G2SAT does not consider the high-precision numerical prediction, which is one of the fundamental challenges in MILP instance generation. In this paper, we propose G2MILP—the first deep generative framework designed for general MILP instances—and we hope to open up a new research direction for the research community.

2.4 MAE and VAE

In this work, we propose a novel masked variational autoencoder paradigm, which draws inspiration from the variational autoencoder (VAE) [22] and the masked autoencoder (MAE) [21]. VAE is a fundamental generative model. However, generic VAE methods generate samples from scratch, which is not suitable in the context of MILP generation, as it may result in plausible instances unaware of the intrinsic combinatorial structures. Therefore, we introduce the masking process into traditional VAE theories. Additionally, we develop a tailored decoder with multiple modules that collaboratively handle MILPs, enabling the application of VAE in this specific domain. Our masked VAE also differs from MAE, which is an auto-regression method that learns to reconstruct the masked images for representation learning. The masked VAE, in contrast, serves as a generative model aiming to generate diverse instances from a masked input. This distinction leads to quite different model structures. For example, the masked VAE incorporates a resample layer and a prior loss during training, and operates in a decoder-only manner during inference, distinguishing it from MAE.

3 METHODOLOGY

In this section, we present our G2MILP framework. First, in Section 3.1, we describe the approach to representing MILP instances as bipartite graphs. Then, in Section 3.2, we focus on the realistic instance generation task and present the basic G2MILP model. Finally, in Section 3.3, we extend G2MILP to the hard instance generation task by proposing the iterative augmenting scheme. The code is released at https://github.com/MIRALab-USTC/L2O-G2MILP.

3.1 Data Representation

A mixed-integer linear programming (MILP) problem takes the form of:

$$\min_{x \in \mathbb{R}^n} \mathbf{c}^\top \mathbf{x},$$

s.t. $A \mathbf{x} \leq \mathbf{b}$,

$l \leq \mathbf{x} \leq u$,

$x_j \in \mathbb{Z}, \forall j \in \mathcal{I},$

where $\mathbf{c} \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, $l \in (\mathbb{R} \cup \{-\infty\})^n$, $u \in (\mathbb{R} \cup \{+\infty\})^n$, and the index set $\mathcal{I} \subset \{1, 2, \ldots, n\}$ includes those indices $j$ where $x_j$ is constrained to be an integer.

To represent each MILP instance, we construct a weighted bipartite graph $G = (\mathcal{V} \cup \mathcal{W}, \mathcal{E})$ as follows [19], [29].

- The constraint vertex set $\mathcal{V} = \{v_1, \ldots, v_m\}$, where each $v_i$ corresponds to the $i$th constraint in Equation 1. The vertex feature $v_i$ of $v_i$ is described by the bias term, i.e., $v_i = (b_i)$.
- The variable vertex set $\mathcal{W} = \{w_1, \ldots, w_n\}$, where each $w_j$ corresponds to the $j$th variable in Equation 1. The vertex feature $w_j$ of $w_j$ is a 9-dimensional vector that contains information of the objective coefficient $c_j$, the variable type, and the bounds $l_j, u_j$.
- The edge set $\mathcal{E} = \{e_{ij}\}$, where an edge $e_{ij}$ connects a constraint vertex $v_i \in \mathcal{V}$ and a variable vertex $w_j \in \mathcal{W}$. The edge feature $e_{ij}$ is described by the coefficient, i.e., $e_{ij} = (a_{ij})$, and there is no edge between $v_i$ and $w_j$ if $a_{ij} = 0$.

As described above, each MILP instance is represented as a weighted bipartite graph, equipped with a tuple of feature matrices $(\mathbf{V}, \mathbf{W}, \mathbf{E})$, where $\mathbf{V}$, $\mathbf{W}$, $\mathbf{E}$ denote stacks.
of vertex features $v_i$, $w_j$ and edge features $e_{ij}$, respectively. The descriptions of these features can be found in Table 1. Such a representation contains all information of the original MILP instance [19]. Notice that, to ensure consistency, we standardize each instance to the form of Equation 1. However, we do not perform data normalization in order to preserve the potential information related to the problem domain in the original formulation. We use the off-the-shelf observation function\(^1\) provided by Ecole [45] to build the bipartite graph MILP instances. We then apply a graph neural network (GNN) to obtain the node representations $h_i$ for all vertices. Then four modules work cooperatively to reconstruct the original graph $G$ based on the node features and the latent vectors. They sequentially determine (a) the bias terms, (b) the degrees, (c) the logits, and (d) the weights. During inference, the model performs in a decoder-only way, and we draw the latent vectors from a standard Gaussian distribution to introduce randomness. We repeat the above mask-and-generate process several times so as to produce new instances.

3.2 Realistic MILP Instance Generation

This section presents the basic G2MILP model, which aims to generate realistic MILP instances. In Section 3.2.1, we derive the masked variational autoencoder (VAE) generative paradigm. In Section 3.2.2, we provide details on the implementation of the model framework. Finally, in Section 3.2.3, we explain the training and inference processes. The model overview is in Figure 2. More implementation details can be found in Appendix A.

1. For additional details on the observation function, readers can consult the following link: https://doc.ecole.ai/py/en/stable/reference/observations.html#ecole.observation.MilpBipartite.

3.2.1 Masked VAE Paradigm

We first introduce our proposed masked VAE paradigm. For the ease of understanding, we provide an intuitive explanation here, and delay the mathematical derivation to Appendix A.1.

Given a graph $G$ drawn from a dataset $D$, we corrupt it through a masking process, denoted by $\tilde{G} \sim \tilde{p}(\tilde{G}|G)$. We aim to build a parameterized generator $p_\theta(G|\tilde{G})$ that can generate new instances $\tilde{G}$ from the corrupted graph $\tilde{G}$. We train the generator by maximizing the log-likelihood $\log p_\theta(\tilde{G}|G) = \log p_\theta(\tilde{G}=G|\tilde{G})$ of reconstructing $\tilde{G}$ given $G$. Therefore, the optimization objective is:

$$\arg \max_\theta \mathbb{E}_{\tilde{G} \sim D} \mathbb{E}_{\tilde{p}(\tilde{G}|G)} \log p_\theta(\tilde{G}|G).$$

To model the randomness in the generation process and produce diverse instances, we follow the standard VAE framework [22], [23] to introduce a latent variable $Z = (z_{v_1}, \cdots, z_{w_m}, z_{w_1}, \cdots, z_{w_n})$, which contains the latent vectors for all vertices. During training, the latent vectors are sampled from a posterior distribution given by a parameterized encoder $q_\phi$, while during inference, they are independently sampled from a prior distribution such as a standard Gaussian distribution. The decoder $p_\theta$ in the masked VAE framework generates new instances from the masked graph $\tilde{G}$ as well as the sampled latent variable $Z$. 

---

Fig. 2. Overview of G2MILP. (a) Masking Process $\tilde{p}(\tilde{G}|G)$. Given a MILP instance, which is represented as a bipartite graph $G$, we randomly label a constraint vertex $\bar{a}$ as [mask] to obtain the masked graph $\tilde{G}$. (b) Encoder $q_\phi(Z|G)$. The encoder is GNN followed by two networks, $\mu_\phi$ and $\Sigma_\phi$, for resampling. During training, we use the encoder to obtain the latent vectors $z_{v_i}$ and $z_{w_j}$ for all vertices. (c) Decoder $p_\theta(G|\tilde{G}, Z)$. We use GNN to obtain the node features $h_i$ and $h_j$, then four modules work cooperatively to reconstruct the original graph $G$ based on the node features and the latent vectors. They sequentially determine $\hat{\delta}$ the degrees, $\hat{\delta}$ the logits, $\hat{\sigma}$ the weights, and $\hat{\Sigma}$ the latent variable constraints. (d) Decoder $p_\theta(G, Z)$. We use $GNN$, also denoted as $\hat{\Sigma}$. The decoder performs in a decoder-only way, and we draw the latent vectors from a standard Gaussian distribution to introduce randomness. We repeat the above mask-and-generate process several times so as to produce new instances.
The evidence lower bound (ELBO), also known as the variational lower bound, is a lower bound estimator of the log-likelihood, and is what we actually optimize during training, because it is more tractable. We can derive the ELBO as:

$$\log p_\theta(\mathcal{G}, \mathcal{Z}) \geq \mathbb{E}_{\mathcal{Z} \sim q_{\phi}(\mathcal{Z}|\mathcal{G})} \left[ \log p_\theta(\mathcal{G}, \mathcal{Z}) \right] - D_{\text{KL}}[q_\phi(\mathcal{Z}|\mathcal{G})||p_\theta(\mathcal{Z})],$$

(3)

where $p_\theta(\mathcal{Z})$ is the prior distribution of $\mathcal{Z}$ and is usually taken as the standard Gaussian $\mathcal{N}(0, I)$, and $D_{\text{KL}}[\cdot || \cdot]$ denotes the KL divergence. Therefore, we formulate the loss function as:

$$L(\theta, \phi) = E_{\mathcal{G} \sim D}[E_{\mathcal{Z} \sim q_{\phi}(\mathcal{Z}|\mathcal{G})} \left[ L(\theta, \phi|\mathcal{G}, \mathcal{Z}) \right] - \beta \cdot D_{\text{KL}}[q_\phi(\mathcal{Z}|\mathcal{G})||\mathcal{N}(0, I)].$$

(4)

In Equation 5: (1) the first term $L_{\text{rec}}$ is the reconstruction loss, which urges the decoder to rebuild the input data according to the masked data and the latent variable. (2) The second term $L_{\text{prior}}$ is used to regularize the posterior distribution in the latent space to approach a standard Gaussian distribution, so that we can sample $\mathcal{Z}$ from the distribution when inference. (3) $\beta$ is a hyperparameter to control the weight of regularization, which is useful in training a VAE model [46].

3.2.2 Model Implementation

To implement Equation 5, we need to instantiate the masking process $\hat{p}(\mathcal{G}|\mathcal{G})$, the encoder $q_\phi(\mathcal{Z}|\mathcal{G})$, and the decoder $p_\theta(\mathcal{G}|\mathcal{G}, \mathcal{Z})$, respectively.

Masking Process. We randomly select a constraint vertex $\tilde{v} \in V$ and mask it and its adjacent edges. We keep the variable vertices unchanged for simplicity. Specifically, we label the vertex $\tilde{v}$ with a special [mask] token, and add virtual edges that link $\tilde{v}$ with each variable vertex. The vertex $\tilde{v}$ and the virtual edges are assigned special embeddings to distinguish them from the others.

Encoder. The encoder $q_\phi(\mathcal{Z}|\mathcal{G})$ is implemented as:

$$q_\phi(\mathcal{Z}|\mathcal{G}) = \prod_{u \in V \setminus W} q_\phi(\mathcal{z}_u|\mathcal{G}),$$

$$q_\phi(\mathcal{z}_u|\mathcal{G}) = \mathcal{N}(\mu_\phi(h_v^u), \exp \Sigma_\phi(h_v^u)),$$

(6)

where $h_v^u$ is the node representation of $u$ obtained by a GNN$_\phi$. $\mathcal{N}$ denotes the Gaussian distribution, and $\mu$ and $\Sigma$ output the mean and the log variance, respectively.

Decoder. The decoder $p_\theta$ reconstructs $\mathcal{G}$ from the corrupted $\mathcal{G}$ and the latent variable $\mathcal{Z}$. We apply a GNN$_\theta$ to obtain the node representations $h_v^u$, denoted as $h_v$ for simplicity. To rebuild the masked constraint vertex $\tilde{v}$, the decoder sequentially determines: (i) the bias $b_{\tilde{v}}$ (i.e., the right-hand side of the constraint), (ii) the degree $d_{\tilde{v}}$ of $\tilde{v}$ (i.e., the number of variables involved in the constraint), (iii) the logits $\hat{d}_{\tilde{v}, u}$ for all variable vertices $u$ to indicate whether they are connected with $\tilde{v}$ (i.e., whether the variables are in the constraint), and (iv) the weights $c_{\tilde{v}, u}$ of the edges (i.e., the coefficients of the variables in the constraint). The decoder is then formulated as:

$$p_\theta(\mathcal{G}|\mathcal{G}, \mathcal{Z}) = p_\theta(b_{\tilde{v}}|\mathcal{G}, \mathcal{Z}) \cdot p_\theta(d_{\tilde{v}}|\mathcal{G}, \mathcal{Z}) \cdot \prod_{u \in W} p_\theta(\hat{d}_{\tilde{v}, u}|\mathcal{G}, \mathcal{Z}, d_{\tilde{v}}).$$

(7)

Therefore, we implement the decoder as four cooperative modules: ① Bias Predictor, ② Degree Predictor, ③ Logs Predictor, and ④ Weights Predictor.

① Bias Predictor. For effective prediction, we incorporate the prior of simple statistics of the dataset—the minimum $\underline{b}$ and the maximum $\overline{b}$ of the bias terms that occur in the dataset—into the predictor. Specifically, we normalize the bias $b_v$ to [0, 1] via $b_v^* = (b_v - \underline{b})/(\overline{b} - \underline{b})$. To predict $b_v^*$, we use an MLP that takes the node representation $h_v$ and the latent vector $z_v$ of $\tilde{v}$ as inputs:

$$\hat{b}_v^* = \sigma \left( \text{MLP}^{\text{bias}}([h_v, z_v]) \right),$$

(8)

where $\sigma(\cdot)$ is the sigmoid function used to restrict the outputs. We use the mean squared error (MSE) loss to train the predictor. At inference time, we apply the inverse transformation to obtain the predicted bias values: $\hat{b}_v = \underline{b} + (\overline{b} - \underline{b}) \cdot \hat{b}_v^*.$

② Degree Predictor. We find that the constraint degrees are crucial to the graph structures and significantly affect the combinatorial properties. Therefore, we use the Degree Predictor to determine coarse-grained degree structure, and then use the Logs Predictor to determine the fine-grained connection details. Similarly to Bias Predictor, we normalize the degree $d_v$ to $d_v^* = (d_v - \underline{d})/(\overline{d} - \underline{d})$, where $\underline{d}$ and $\overline{d}$ are the minimum and maximum degrees in the dataset, respectively. We use an MLP to predict $d_v^*$:

$$\hat{d}_v^* = \sigma \left( \text{MLP}^{\text{deg}}([h_v, z_v]) \right).$$

(9)

We use MSE loss for training, and round the predicted degree to the nearest integer $d_v$ for inference.

③ Logs Predictor. To predict the logits $\hat{d}_{\tilde{v}, u}$ indicating whether a variable vertex $u \in W$ is connected with $\tilde{v}$, we use one MLP that takes the representation $h_u$ and the latent vector $z_u$ of $u$ as inputs:

$$\hat{d}_{\tilde{v}, u} = \sigma \left( \text{MLP}^{\text{logs}}([h_u, z_u]) \right).$$

(10)

We use binary cross-entropy (BCE) loss to train the logistical regression module. As positive samples (i.e., variables connected with a constraint) are often scarce, we use one negative sample for each positive sample during training. The loss function is:

$$L_{\text{logs}} = -E_{(\hat{e}, u) \sim p_{\text{pos}}} \left[ \log (\hat{d}_{\tilde{v}, u}) \right] - E_{(\hat{e}, u) \sim p_{\text{neg}}} \left[ \log (1 - \hat{d}_{\tilde{v}, u}) \right],$$

(11)

where $p_{\text{pos}}$ and $p_{\text{neg}}$ denote the distributions over positive and negative samples, respectively. At inference time, we connect $d_{\tilde{v}}$ variable vertices with the top logits to $\tilde{v}$, i.e.,

$$\hat{d}_{\tilde{v}, u} = \begin{cases} 1, & u \in \text{arg TopK} \{\hat{d}_{\tilde{v}, u} | u \in W\}, \\ 0, & \text{otherwise}. \end{cases}$$

(12)

2. Notation-wise, we use $\hat{x}$ to denote the predicted variable in $\mathcal{G}$ that corresponds to $x$ in $\hat{G}$.
The training and inference procedures are similar to those of
four prediction tasks, respectively, and \(\alpha\) reconstruction loss, is written as
above, the decoder involves four modules, each optimized
from the masked graph. During training, we use the original graph and replace it with a generated one. We define a hyperparameter \(\eta\) to adjust the ratio of iterations to the number of constraints, i.e., \(N_{\text{iters}} = \eta \cdot |V|\). Naturally, a larger value of \(\eta\) results in instances that are more novel, while a smaller value of \(\eta\) yields instances that exhibit better similarity to the training set. We describe the training and inference procedures in Algorithm 1 and Algorithm 2, respectively.

3.3 Hard MILP Instance Generation
Following [20], we use the solving time of an instance when solved by a MILP solver, such as Gurobi, as a metric to measure its computational hardness. We propose a hardness-oriented iterative augmenting scheme to guide the generator to generate hard MILPs effectively. It involves an iterative process where we utilize the current generator to produce new instances and subsequently fine-tune the generator using the most challenging instances that we have discovered. The overview of our proposed hardness-oriented iterative augmenting scheme is in Figure 3.

Specifically, the scheme initiates with a given dataset \(D\) of MILP instances. We employ this dataset to pretrain a generator \(p_{\theta}\) following the steps in Algorithm 1, so that the model can generate instances similar to those in the training set. We use the training set to initialize a storage \(S\) to maintain a record of the most challenging MILP instances that we have discovered so far. Within each iteration, the following steps are performed.

1) The generator \(p_{\theta}\) samples \(N\) new instances to extend the storage \(S\).
2) We update \(S\) by selecting the \(K\) instances that require the most solving time for the solver.
3) We fine-tune the generator \(p_{\theta}\) using the current instances in \(S\).

The detailed procedure is in Algorithm 3.

\[\%\text{Weights Predictor.}\] Finally, we use an MLP to predict the normalized weights \(\hat{e}_{\bar{v}, u}^{*}\) for nodes \(u \in \bar{V}\) that are connected with \(\bar{v}\):

\[
\hat{e}_{\bar{v}, u}^{*} = \sigma \left( \text{MLP}_{\theta}^{\text{weights}} ([h_{u}, z_{u}]) \right) . \quad (13)
\]

The training and inference procedures are similar to those of Bias Predictor.

3.2.3 Training and Inference
During training, we use the original graph \(G\) to provide supervision signals for the decoder, guiding it to reconstruct \(G\) from the masked \(\tilde{G}\) and the encoded \(Z\). As described above, the decoder involves four modules, each optimized by a prediction task. The first term in Equation 5, i.e., the reconstruction loss, is written as

\[
L_{\text{rec}} = E_{G \sim D, \tilde{G} \sim \tilde{p}(\tilde{G}|G)} \left[ \sum_{i=1}^{4} \alpha_i \cdot L_i(\theta, \phi|\tilde{G}, \tilde{G}) \right] , \quad (14)
\]

where \(L_i(\theta, \phi|\tilde{G}, \tilde{G})(i = 1, 2, 3, 4)\) are loss functions for the four prediction tasks, respectively, and \(\alpha_i\) are hyperparameters.

\[\%\text{Weights Predictor.}\] Finally, we use an MLP to predict the normalized weights \(\hat{e}_{\bar{v}, u}^{*}\) for nodes \(u \in \bar{V}\) that are connected with \(\bar{v}\):

\[
\hat{e}_{\bar{v}, u}^{*} = \sigma \left( \text{MLP}_{\theta}^{\text{weights}} ([h_{u}, z_{u}]) \right) . \quad (13)
\]

The training and inference procedures are similar to those of Bias Predictor.

3.2.3 Training and Inference
During training, we use the original graph \(G\) to provide supervision signals for the decoder, guiding it to reconstruct \(G\) from the masked \(\tilde{G}\) and the encoded \(Z\). As described above, the decoder involves four modules, each optimized by a prediction task. The first term in Equation 5, i.e., the reconstruction loss, is written as

\[
L_{\text{rec}} = E_{G \sim D, \tilde{G} \sim \tilde{p}(\tilde{G}|G)} \left[ \sum_{i=1}^{4} \alpha_i \cdot L_i(\theta, \phi|\tilde{G}, \tilde{G}) \right] , \quad (14)
\]

where \(L_i(\theta, \phi|\tilde{G}, \tilde{G})(i = 1, 2, 3, 4)\) are loss functions for the four prediction tasks, respectively, and \(\alpha_i\) are hyperparameters.
We conduct extensive experiments to demonstrate the effectiveness of our model. More experimental details can be found in Appendix B.

4.1 Setup

We consider two task settings: realistic MILP instance generation and hard MILP instance generation.

Realistic MILP Instance Generation. To evaluate the quality of the generated MILP instances, we design three benchmarks so as to answer the following research questions. (1) How well can the generated instances preserve the graph structures of the training set? (2) How closely do the generated instances resemble the computational hardness of real-world instances? (3) How effectively do they facilitate downstream tasks to improve solver performance?

I. Structural Distributional Similarity. We consider 11 classical statistics, which are shown in Table 2, as descriptors to represent features of the instances [18], [47]. In line with a widely used graph generation benchmark [48], we calculate the statistics for each instance, and then compute the Jensen-Shannon (JS) divergence [49] $D_{\text{JS},i}$ between the generated samples and the training set for each descriptor $i \in \{1, \ldots, 11\}$. To compute the JS divergence, we estimate the distributions using the histogram function in numpy and the cross entropy using the entropy function in scipy. The JS divergence falls in the range $[0, \log 2]$, so we standardize it into a similarity score $s_i$ via:

$$s_i = \frac{1}{\log 2} (\log 2 - D_{\text{JS},i}),$$

Then we compute the mean of the 11 scores for the descriptors to obtain the final score $s$:

$$s = \frac{1}{11} \sum_{i=1}^{11} s_i.$$

Hence the final score ranges from 0 to 1, with a higher score indicating better similarity. Notice that for a fair comparison, we exclude statistics that remain constant in our approach, such as problem size and objective coefficients. We implement another version of metric that involves more statistics, and the results are in Appendix C.3.

II. Computational Hardness. The computational hardness of instances is another critical metric to assess the quality of the generated instances. We draw an analogy from the SAT generation community, where though many progresses achieved, it is widely acknowledged that the generated SAT instances differ significantly from real-world ones in the computational hardness [50], and this issue remains inadequately addressed. In our work, we make efforts to mitigate this problem, even in the context of MILP generation, a more challenging task. To this end, we leverage the state-of-the-art solver Gurobi [5] to solve the instances, and we report the solving time and the numbers of branching nodes during the solving process, which can directly reflect the computational hardness of instances [20].

III. Downstream Tasks. We consider two downstream tasks to examine the potential benefits of the generated instances in practical applications. We employ G2MILP to generate new instances and augment the original datasets, and then evaluate whether the enriched datasets can improve the performance of the downstream tasks. The considered tasks include predicting the optimal values of the MILP problem, as discussed in Chen et al. [19], and applying a predict-and-search framework for solving MILPs, as proposed by Han et al. [31].

Hard MILP Instance Generation. We measure the hardness of the MILP instances by analyzing the solving time achieved using Gurobi. In this research task, our objective is to construct a collection of difficult MILP instances, starting from a set of real-world examples. We employ the proposed hardness-oriented iterative augmenting scheme to achieve this. Instead of solely focusing on identifying the single most challenging instance, our approach is superior in producing a diverse set of hard instances. To demonstrate this, we sort

\begin{table}[h]
\centering
\caption{The descriptions of statistics used for measuring the structural distributional similarity. These statistics are calculated on the bipartite graph extracted by Ecole.}
\begin{tabular}{|l|l|}
\hline
Feature & Description \\
\hline
\hline
\hline
\text{coef\_dens} & Fraction of non-zero entries in $A$, i.e., $|\mathcal{E}|/(|\mathcal{V}| \cdot |\mathcal{V}|)$. \\
\hline
\text{cons\_degree\_mean} & Mean degree of constraint vertices in $\mathcal{V}$. \\
\hline
\text{cons\_degree\_std} & Std of degrees of constraint vertices in $\mathcal{V}$. \\
\hline
\text{var\_degree\_mean} & Mean degree of variable vertices in $\mathcal{W}$. \\
\hline
\text{var\_degree\_std} & Std of degrees of variance vertices in $\mathcal{W}$. \\
\hline
\text{lhs\_mean} & Mean of non-zero entries in $A$. \\
\hline
\text{lhs\_std} & Std of non-zero entries in $A$. \\
\hline
\text{rhs\_mean} & Mean of $b$. \\
\hline
\text{rhs\_std} & Std of $b$. \\
\hline
\text{clustering\_coef} & Clustering coefficient of the graph. \\
\hline
\text{modularity} & Modularity of the graph. \\
\hline
\end{tabular}
\end{table}
the obtained instances based on their solving time using Gurobi, arranging them in descending order. Subsequently, we examine the top 1, top 10, and top 50 instances with the highest solving times, and we calculate the average solving time for these three collections of instances, respectively.

4.1.2 Datasets
We consider four different datasets of various sizes. (1) Large datasets. We evaluate the model’s capability of learning data distributions using two well-known synthetic MILP datasets: Maximum Independent Set (MIS) [51] and Set Covering [16]. We follow previous works [11], [12] to artificially generate 1000 instances for each of them. (2) Medium dataset. Mixed-integer Knapsack (MIK) is a widely used dataset [34], which consists of 80 training instances and 10 test instances. We use this dataset to evaluate the model’s performance both on the distribution learning benchmarks and the downstream task. (3) Small dataset. We construct a small subset of MIPLIB 2017 [13] by collecting a group of problems called Nurse Scheduling problems. This dataset comes from real-world scenarios and consists of only 4 instances, 2 for training and 2 for test, respectively. Since the statistics are meaningless for such an extremely small dataset, we use it only to demonstrate the effectiveness of generated instances in facilitating downstream tasks.

4.1.3 Baselines
G2MILP is the first deep learning generative framework for MILP instances, and thus, we do not have any learning-based models for comparison purpose. Therefore, we compare G2MILP with a heuristic MILP instance generator, namely Bowly [17]. Bowly can create feasible and bounded MILP instances while controlling some specific statistical features such as coefficient density and coefficient mean. We set all the controllable parameters to match the corresponding statistics of the training set, allowing Bowly to imitate the training set to some extent. We also consider a useful baseline, namely Random, to demonstrate the effectiveness of deep neural networks in G2MILP. Random employs the same generation procedure as G2MILP, but replaces all neural networks in the decoder with random generators. We set the masking ratio $\eta$ for Random and G2MILP to 0.01, 0.05, and 0.1 to show how this hyperparameter helps balance the novelty and similarity.

4.2 Results: Realistic MILP Instance Generation
4.2.1 Structural Distributional Similarity
We present the structural distributional similarity scores between each pair of datasets in Table 3. The results indicate that our designed metric is reasonable in the sense that datasets obtain high scores with themselves and low scores with different domains. Table 4 shows the similarity scores between generated instances and the corresponding training sets. We generate 1000 instances for each dataset to compute the similarity scores. The results suggest that G2MILP closely fits the data distribution, while Bowly, which relies on heuristic rules to control the statistical features, falls short of our expectations. Furthermore, we observe that G2MILP outperforms Random, indicating that deep learning contributes to the model’s performance. As expected, a higher masking ratio $\eta$ results in generating more novel instances but reduces their similarity to the training sets.

4.2.2 Computational Hardness
We report the average solving time and numbers of branching nodes in Table 5 and Table 6, respectively. The results indicate that instances generated by Bowly are relatively easy, and the hardness of those generated by Random is inconclusive. In contrast, G2MILP is capable of preserving the computational hardness of the original training sets. Notably, even without imposing rules to guarantee the feasibility and boundedness of generated instances, G2MILP automatically learns from the data and produces feasible and bounded instances.

4.2.3 Downstream Tasks
Optimal Value Prediction. Two datasets, MIK and Nurse Scheduling, are considered, with medium and extremely small sizes, respectively. Following [19], we employ a GNN as a predictive model. The GNN structure is similar to the GNNs in G2MILP. We obtain the graph representation using mean pooling over all vertices, followed by a two-layer MLP to predict the optimal values of the instances. For each dataset, we train a GNN predictive model on the training set. Specifically, for MIK, we use 80% of instances for training, 20% of instances for validating, and train for 1000 epochs to select the best checkpoint based on validation MSE. For Nurse Scheduling, we use both instances to train the model for 80 epochs. We use the generative models, Bowly, Random, and G2MILP, to generate 20 instances similar to the training sets. For Random and G2MILP, we mix together the instances generated by setting the masking ratio $\eta$ to 0.01 and 0.05, respectively. Next, we use the generated instances to enrich the original training sets, and use the enriched data to train another predictive model. We test all the trained model on

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Structural similarity scores between each pair of datasets. Higher is better.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIS</td>
</tr>
<tr>
<td>MIS</td>
<td>0.998</td>
</tr>
<tr>
<td>SetCover</td>
<td>-</td>
</tr>
<tr>
<td>MIK</td>
<td>-</td>
</tr>
</tbody>
</table>

| Table 4 | Structural distributional similarity scores between the generated instances with the training datasets. Higher is better. $\eta$ is the masking ratio. We do not report the results of Bowly on MIK because Ecole [45] and SCIP [6] fail to read the generated instances due to large numerical values. |
|---------|---------|---------|--------|
|         | MIS     | SetCover | MIK    |
| Bowly   | 0.184   | 0.197   | -      |
| $\eta = 0.01$ | Random | 0.651   | 0.735  | 0.969  |
| G2MILP  | 0.997   | 0.835   | 0.991  |
| $\eta = 0.05$ | Random | 0.580   | 0.613  | 0.840  |
| G2MILP  | 0.940   | 0.782   | 0.953  |
| $\eta = 0.1$ | Random | 0.512   | 0.556  | 0.700  |
| G2MILP  | 0.895   | 0.782   | 0.918  |
Fig. 4. Results of the optimal value prediction task. Bars indicate the relative MSE to the model trained on the original training sets, and lines represent the relative performance improvement.

Table 8 Results of the predict-and-search framework on MIS. The training set contains 100 instances, and we generate 100 new instances. For Random and G2MILP, masking ratio is 0.01. Time means the time for Gurobi to find the optimal solution in the trust region with augmenting data generated by different models. Bowly leads to the framework failing to find optimal solutions in the trust region.

We present the MSE relative to the default model trained on the original training sets in Figure 4. The detailed results are in Table 7. On MIK, instances generated by Bowly result in numerical issues, as some generated coefficients are excessively large, so that Ecole and Gurobi fail to read them. On Nurse Scheduling, Random fails to generate feasible instances, and Bowly yields a minor improvement. Notably, G2MILP is the only method that demonstrates performance improvement on both datasets, reducing the MSE by 73.7% and 24.3%, respectively. Moreover, G2MILP allows for the training of the model with even minimal data.

**Predict-and-Search.** We conduct experiments on a neural solver, i.e., the predict-and-search framework proposed by Han et al. [31]. Specifically, they propose a framework that first predicts a solution and then uses solvers to search for the optimal solutions in a trust region. We consider using generated instances to enhance the predictive model. We first train the predictive model on 100 MIS instances, and then use the generative models to generate 100 new instances to augment the dataset. The results are in Table 8. Bowly generates low-quality data that Disturbs the model training, so that there is no optimal solution in the trust region around the predicted solution. Though both Random and G2MILP can enhance the solving framework to reduce solving time, we can see G2MILP significantly outperforms Random.

### Table 7 Results on the optimal value prediction task (mean±std).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MSE Improvement</th>
<th>Nurse Scheduling Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>0.0236</td>
<td>0.00%</td>
</tr>
<tr>
<td>Bowly</td>
<td>-</td>
<td>±66.53</td>
</tr>
<tr>
<td>Random</td>
<td>0.0104</td>
<td>±6.7%</td>
</tr>
<tr>
<td>G2MILP</td>
<td>±0.0073</td>
<td>±5.9%</td>
</tr>
</tbody>
</table>

### Table 5 Average solving time (s) of instances solved by Gurobi (mean±std). η is the masking ratio. Numbers in the parentheses are relative errors with respect to the training sets (lower is better).

<table>
<thead>
<tr>
<th></th>
<th>MIS</th>
<th>SetCover</th>
<th>MIK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Set</td>
<td>0.349±0.05</td>
<td>2.344±0.13</td>
<td>0.198±0.04</td>
</tr>
<tr>
<td>Bowly</td>
<td>0.007±0.00</td>
<td>0.048±0.00</td>
<td>0.001±0.00</td>
</tr>
<tr>
<td></td>
<td>(97.9%)</td>
<td>(97.9%)</td>
<td>(99.8%)</td>
</tr>
<tr>
<td>η = 0.01</td>
<td>Random</td>
<td>0.311±0.05</td>
<td>2.044±0.19</td>
</tr>
<tr>
<td></td>
<td>(10.8%)</td>
<td>(12.8%)</td>
<td>(96.1%)</td>
</tr>
<tr>
<td></td>
<td>G2MILP</td>
<td>0.354±0.06</td>
<td>2.360±0.18</td>
</tr>
<tr>
<td></td>
<td>(1.5%)</td>
<td>(0.8%)</td>
<td>(14.7%)</td>
</tr>
<tr>
<td>η = 0.05</td>
<td>Random</td>
<td>0.569±0.09</td>
<td>2.010±0.11</td>
</tr>
<tr>
<td></td>
<td>(63.0%)</td>
<td>(14.3%)</td>
<td>(97.9%)</td>
</tr>
<tr>
<td></td>
<td>G2MILP</td>
<td>0.292±0.07</td>
<td>2.533±0.15</td>
</tr>
<tr>
<td></td>
<td>(16.3%)</td>
<td>(8.1%)</td>
<td>(35.1%)</td>
</tr>
<tr>
<td>η = 0.1</td>
<td>Random</td>
<td>2.367±0.35</td>
<td>1.988±0.17</td>
</tr>
<tr>
<td></td>
<td>(578.2%)</td>
<td>(15.2%)</td>
<td>(97.6%)</td>
</tr>
<tr>
<td></td>
<td>G2MILP</td>
<td>0.214±0.05</td>
<td>2.108±0.21</td>
</tr>
<tr>
<td></td>
<td>(38.7%)</td>
<td>(10.0%)</td>
<td>(63.9%)</td>
</tr>
</tbody>
</table>

**Table 6 Average numbers of branching nodes of instances solved by Gurobi. η is the masking ratio. Numbers in the parentheses are relative errors with respect to the training sets (lower is better).**

<table>
<thead>
<tr>
<th></th>
<th>MIS</th>
<th>SetCover</th>
<th>MIK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Set</td>
<td>16.09</td>
<td>838.56</td>
<td>175.35</td>
</tr>
<tr>
<td>Bowly</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>(100.0%)</td>
<td>(100.0%)</td>
<td>(100.0%)</td>
</tr>
<tr>
<td>η = 0.01</td>
<td>Random</td>
<td>20.60</td>
<td>838.51</td>
</tr>
<tr>
<td></td>
<td>(28.1%)</td>
<td>(0.0%)</td>
<td>(99.5%)</td>
</tr>
<tr>
<td></td>
<td>G2MILP</td>
<td>15.03</td>
<td>876.09</td>
</tr>
<tr>
<td></td>
<td>(6.6%)</td>
<td>(4.4%)</td>
<td>(14.7%)</td>
</tr>
<tr>
<td>η = 0.05</td>
<td>Random</td>
<td>76.22</td>
<td>765.30</td>
</tr>
<tr>
<td></td>
<td>(373.7%)</td>
<td>(8.7%)</td>
<td>(100%)</td>
</tr>
<tr>
<td></td>
<td>G2MILP</td>
<td>10.58</td>
<td>874.46</td>
</tr>
<tr>
<td></td>
<td>(34.2%)</td>
<td>(4.3%)</td>
<td>(34.2%)</td>
</tr>
<tr>
<td>η = 0.1</td>
<td>Random</td>
<td>484.47</td>
<td>731.14</td>
</tr>
<tr>
<td></td>
<td>(2911.2%)</td>
<td>(12.8%)</td>
<td>(100%)</td>
</tr>
<tr>
<td></td>
<td>G2MILP</td>
<td>4.61</td>
<td>876.92</td>
</tr>
<tr>
<td></td>
<td>(71.3%)</td>
<td>(4.6%)</td>
<td>(20.1%)</td>
</tr>
</tbody>
</table>

We present the MSE relative to the default model trained on the original training sets in Figure 4. The detailed results are in Table 7. On MIK, instances generated by Bowly result in numerical issues, as some generated coefficients are excessively large, so that Ecole and Gurobi fail to read them. On Nurse Scheduling, Random fails to generate feasible instances, and Bowly yields a minor improvement. Notably, G2MILP is the only method that demonstrates performance improvement on both datasets, reducing the MSE by 73.7% and 24.3%, respectively. Moreover, G2MILP allows for the training of the model with even minimal data.

**Predict-and-Search.** We conduct experiments on a neural solver, i.e., the predict-and-search framework proposed by Han et al. [31]. Specifically, they propose a framework that first predicts a solution and then uses solvers to search for the optimal solutions in a trust region. We consider using generated instances to enhance the predictive model. We first train the predictive model on 100 MIS instances, and then use the generative models to generate 100 new instances to augment the dataset. The results are in Table 8. Bowly generates low-quality data that Disturbs the model training, so that there is no optimal solution in the trust region around the predicted solution. Though both Random and G2MILP can enhance the solving framework to reduce solving time, we can see G2MILP significantly outperforms Random.

### Table 8 Results of the predict-and-search framework on MIS. The training set contains 100 instances, and we generate 100 new instances. For Random and G2MILP, masking ratio is 0.01. Time means the time for Gurobi to find the optimal solution in the trust region with augmenting data generated by different models. Bowly leads to the framework failing to find optimal solutions in the trust region.

<table>
<thead>
<tr>
<th>Method</th>
<th>Training Set</th>
<th>Bowly</th>
<th>Random</th>
<th>G2MILP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0.041</td>
<td>17/100 fail</td>
<td>0.037</td>
<td>0.032</td>
</tr>
<tr>
<td></td>
<td>(±0.006)</td>
<td></td>
<td>(±0.003)</td>
<td>(±0.004)</td>
</tr>
</tbody>
</table>
in Table 4 and Figure 4. From the results we have the following conclusions. (1) though empirically a smaller leads to a relatively better performance, G2MILP maintains a high similarity performance even when \( \eta \) is large. (2) The downstream task performance does not drops significantly. This makes sense because smaller \( \eta \) leads to more similar instances, while larger \( \eta \) leads to more diverse (but still similar) instances, both of which can benefit downstream tasks. (3) G2MILP always outperforms Random, which demonstrates that the learning paradigm helps maintain the performance. (4) Bowly fails on this dataset because its generated instances lead to numerical issues and cannot be read by Gurobi or SCIP. Moreover, in real applications, it is reasonable and flexible to adjust the hyperparameter to achieve good performances in different scenarios. Second, we implement different versions of G2MILP , which enable us to mask and modify either constraints, variables, or both. Third, we investigate different orders of masking constraints, including uniformly sampling and sampling according to the vertex indices. Those results are in Appendix C.2.

Size of Dataset. We conduct experiments on different sizes of the original datasets and different ratio of generated instances to original ones on MIS. Results are in Table 9. The results show that G2MILP yields performance improvements across datasets of varying sizes.

![Image](image_url)

**Fig. 5.** (a) Distributional similarity score (higher is better) and (b) Relative MSE (lower is better) v.s. masking ratio \( \eta \).

**Fig. 6.** The t-SNE visualization of MILP instance representations for MIK. Each point represents an instance. Red points are from the training set and blue points are instances generated by G2MILP.

**Fig. 7.** (a) Average solving time and (b) hardness boost in each iteration step on MIS.

Table 9 Results on the optimal value prediction task on MIS with different dataset sizes. In the table, “#MILPs” denotes the number of instances in the training sets, and “Augment%” denotes the ratio of generated instances to training instances.

<table>
<thead>
<tr>
<th>#MILPs</th>
<th>Augment%</th>
<th>MSE</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>25%</td>
<td>1.014</td>
<td>23.1%</td>
</tr>
<tr>
<td>0</td>
<td>50%</td>
<td>0.998</td>
<td>24.3%</td>
</tr>
<tr>
<td>0</td>
<td>100%</td>
<td>0.982</td>
<td>25.5%</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>25%</td>
<td>0.786</td>
<td>1.5%</td>
</tr>
<tr>
<td>0</td>
<td>50%</td>
<td>0.752</td>
<td>5.8%</td>
</tr>
<tr>
<td>0</td>
<td>100%</td>
<td>0.561</td>
<td>23.7%</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>25%</td>
<td>0.283</td>
<td>19.0%</td>
</tr>
<tr>
<td>0</td>
<td>50%</td>
<td>0.243</td>
<td>17.3%</td>
</tr>
<tr>
<td>0</td>
<td>100%</td>
<td>0.202</td>
<td>31.3%</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>25%</td>
<td>0.168</td>
<td>10.6%</td>
</tr>
<tr>
<td>0</td>
<td>50%</td>
<td>0.175</td>
<td>6.9%</td>
</tr>
<tr>
<td>0</td>
<td>100%</td>
<td>0.170</td>
<td>9.6%</td>
</tr>
</tbody>
</table>

**Table 10** Average solving time (s) of instances solved by Gurobi. Here, “top k” indicates that we calculate the average time over the top k instances. “Dataset” denotes the instances in the original dataset, and “G2MILP” denotes the generated instances in 7 loops.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>G2MILP</th>
<th>Hardness Boost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>top1</td>
<td>4.918</td>
</tr>
<tr>
<td></td>
<td>top10</td>
<td>2.687</td>
</tr>
<tr>
<td></td>
<td>top50</td>
<td>1.37</td>
</tr>
<tr>
<td>SetCover</td>
<td>top1</td>
<td>26.997</td>
</tr>
<tr>
<td></td>
<td>top10</td>
<td>15.994</td>
</tr>
<tr>
<td></td>
<td>top50</td>
<td>8.905</td>
</tr>
<tr>
<td>MIK</td>
<td>top1</td>
<td>0.986</td>
</tr>
<tr>
<td></td>
<td>top10</td>
<td>0.491</td>
</tr>
<tr>
<td></td>
<td>top50</td>
<td>0.275</td>
</tr>
</tbody>
</table>

**Visualization.** We visualize the instance representations for MIK in Figure 6. Specifically, we use the G2MILP encoder to obtain the instance representations, and then apply t-SNE dimensionality reduction [52] for visualization. We observe that the generated instances, while closely resembling the training set, contribute to a broader and more continuous exploration of the problem space, thereby enhancing model robustness and generalization. Additionally, by increasing the masking ratio \( \eta \), we can effectively explore a wider problem space beyond the confines of the training sets.

**4.3 Results: Hard MILP Instance Generation**

To initiate the generation process, we utilize the pretrained G2MILP model, which was trained in the task of realistic MILP instance generation. We simply set the masking ratio \( \eta \) to 0.05. The storage size is set to 100, and we perform 7 iterations to iteratively augment the model. During each
iteration, we generate 200 instances at specific checkpoints during training to identify and retain the best checkpoints. Subsequently, we employ the best model to generate 1000 instances to extend the storage. Table 10 presents the average solving time, determined by Gurobi, for the top 1, top 10, and top 50 instances, respectively. We also report the hardness boost, which is defined as:

\[
\text{Hardness Boost} = \frac{\text{Average Solving Time of Generated Instances}}{\text{Average Solving Time of Instances in Dataset}}
\]

The results demonstrate the effectiveness of G2MILP in generating challenging MILP instances, resulting in a considerable boost in the hardness of the original dataset.

Figure 7 showcases the growth curve of the hardness during the iterative augmentation process, while Figure 8 depicts the distributions of the instances within the storage. The observed boost in 10 iterations is nearly two orders of magnitude. These visualizations demonstrate the growing trends of the hardness of the generated instances, implying that G2MILP has the potential to construct more hard instances with the number of iterations increasing.

5 LIMITATIONS, FUTURE AVENUES, AND CONCLUSIONS

5.1 Limitations
First, in this paper, we develop G2MILP by iteratively corrupting and replacing the constraints vertices. More versatile masking schemes can be explored. Second, employing more sophisticated designs would enable us to control critical properties such as sizes of the instances. Third, the hardness-oriented iterative augmenting scheme relies on numerous calls of a MILP solver, which is time-consuming. It will be meaningful to devise more surrogate metrics for measuring the harness of instances, thus reducing the cost of solving these instances.

5.2 Future Avenues
In addition to producing realistic or challenging instances, this research has many other promising technical implications. First, such a generator will assist researchers to gain insights into different data domains and the explored space of MILP instances. For example, studying the process of generating hard instances will help better understand what affects the hardness of MILPs. Second, based on a generative model, we can establish an adversarial framework, where the generator aims to identify challenging cases for the solver, thus automatically enhancing the solver’s ability to handle complex scenarios. Third, training a generative model involves learning the data distribution and deriving representations through unsupervised learning. Consequently, it can serve as a pre-trained model and can benefit downstream tasks across various domains. Though the aforementioned routes are out of the scope of this paper, we believe that this paper serves as an entrance for them, and we expect further efforts in this field.

5.3 Conclusions
In this paper, we propose G2MILP, which to the best of our knowledge is the first deep generative framework for MILP instances. We demonstrate the effectiveness of G2MILP in both generating realistic MILP instances and generating hard MILP instances. This work opens up new avenues for research on DL-based MILP instance generative models.

ACKNOWLEDGMENTS
The authors would like to thank all the anonymous reviewers for their insightful comments. This work was supported in part by National Key R&D Program of China under contract 2022ZD0119801, National Nature Science Foundations of China grants U19B2026, U19B2044, 61836011, 62021001, and 61836006.
REFERENCES


Jie Wang received the B.Sc. degree in electronic information science and technology from University of Science and Technology of China, Hefei, China, in 2005, and the Ph.D. degree in computational science from the Florida State University, Tallahassee, FL, in 2011. He is currently a professor in the Department of Electronic Engineering and Information Science at University of Science and Technology of China, Hefei, China. His research interests include reinforcement learning, knowledge graph, large-scale optimization, deep learning, etc. He is a senior member of IEEE.

Zijie Geng received the B.Sc. degree in mathematics and applied mathematics from School of the Gifted Young, University of Science and Technology of China, Hefei, China, in 2022. He is currently an M.Sc. candidate in the Department of Electronic Engineering and Information Science at University of Science and Technology of China, Hefei, China. His research interests include reinforcement learning, graph learning, AI for science and machine learning for combinatorial optimization.

Xijun Li is a senior researcher of HUAWEI Noah’s Ark Lab. Before that, he has received M.Sc. degree from Shanghai Jiao Tong University, P.R. China, in 2018. He is working towards his Ph.D. degree in the University of Science and Technology of China (HUAWEI-USTC Joint Ph.D. Program) under the supervision of Prof. Jie Wang. He has published 10+ papers on top peer-reviewed conferences and journals (such as ICLR, KDD, ICDE, SIGMOD, DAC, CIKM, ICCDCS, TCYB, etc.) and applied/published 12 patents (first author in 6 patents) with Noah’s Ark Lab. And he has won the championship of student leaderboard in Dual Track of NeurIPS’21 ML4CO Competition. His recent research interests focus on Mathematical Programming Solver, Learning to Optimization (L2O) and Machine Learning for Computer System (ML4CS).

Jianye Hao received the B.S. degree in computer science from Harbin Institute of Technology, Harbin, China, in 2008, and the Ph.D. degree in computer science and engineering from The Chinese University of Hong Kong, Hong Kong, in 2013. He was a Post-Doctoral Fellow with the Massachusetts Institute of Technology, Cambridge, MA, USA, and the Singapore University of Technology and Design, Singapore. He is currently the Director of the Decision Making and Reasoning Laboratory, Noah’s Ark Laboratory, Huawei, Beijing, and an Associate Professor with Tianjin University, Tianjin, China. He has authored or coauthored over 100 papers on artificial intelligence conferences and journals. His research interests include deep reinforcement learning and multi-agent systems. Dr. Hao has received the Best Paper Award of ASEE2019, DAII2019, and CoRL2020.

Yongdong Zhang received the Ph.D. degree in electronic engineering from Xidian University in 1992, and the M.S. and Ph.D. degrees in computer science from the Harbin Institute of Technology in 1996 and 1999, respectively. He is currently a Professor with the University of Science and Technology of China, where he is also the Dean of the School of Information Science and Technology. Before that, he was a Principal Researcher and the Research Manager with Microsoft Research Asia. His research interests include image and video compression, media communication, and media analysis and synthesis. He has authored or coauthored over 200 high quality articles (including several dozens of IEEE Transaction papers) and top conference papers on MOBICOM, SIGIR, CVPR, and ACM MM. He has 77 granted U.S. patents. His 15 techniques have been adopted into international video coding standards. As a coauthor, he received the Best Paper Award at 2009 IEEE Transactions on Circuits and Systems for Video Technology, PCM 2008, and SPIE VCIP 2007. He also received the Best Associate Editor Award from IEEE Circuits and Systems Society in 2012. He also serves as the TPC Chair for MMSP 2011, VCIP 2010, and PCM 2009, and the Special Sessions Chair for ICME 2010 and ISCAS 2013. He serves as an Associate Editor for IEEE Transactions on Circuits and Systems for Video Technology, IEEE Transactions ON Multimedia, and several other international journals.

Feng Wu received the B.S. degree in electrical engineering from Xidian University in 1992, and the M.S. and Ph.D. degrees in computer science from the Harbin Institute of Technology in 1996 and 1999, respectively. He is currently a Professor with the University of Science and Technology of China, Hefei, China. His research interests include multimedia content security, video encoding, and streaming media technology. He serves as an Editorial Board Member of Multimedia Systems journal and Neurocomputing. He was the recipient of the Best Paper Award in PCM 2013, ICIMCS 2013, and ICME 2010, and the Best Paper Candidate in ICME 2011. He is a Senior Member of IEEE.
A.1 The Derivation of Masked Variational Auto-Encoder

We consider a variable with a distribution $p(x)$. We draw samples from this distribution and apply a masking process to transform each sample $x$ into $\hat{x}$ through a given probability $\hat{p}(\hat{x}|x)$. Our objective is to construct a parameterized generator $p_\theta(x|\hat{x})$ to produce new data based on the masked data $\hat{x}$. We assume that the generation process involves an unobserved continuous random variable $z$ that is independent of $\hat{x}$, i.e., $z \perp \hat{x}$. Consequently, we obtain the following equation:

$$p_\theta(x|\hat{x}) = \frac{p_\theta(x,z,\hat{x})}{p_\theta(z,\hat{x})} = \frac{p_\theta(x,z,\hat{x})p_\theta(z)}{p_\theta(z,\hat{x})}. \quad (19)$$

We introduce a probabilistic encoder $q_\phi(z|x)$ for approximating the intractable latent variable distribution. We can then derive the follows:

$$\log p_\theta(x|\hat{x}) = \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log p_\theta(x|z,\hat{x}) \right]$$

$$= \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \frac{p_\theta(x,z,\hat{x})}{p_\theta(z,\hat{x})} \right]$$

$$= \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \frac{p_\theta(x,z,\hat{x})p_\theta(z)}{p_\theta(z,\hat{x})} \right]$$

$$= \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \frac{p_\theta(x,z,\hat{x})}{p_\theta(z,\hat{x})} \right] + \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \frac{q_\phi(z|x)}{p_\theta(z,\hat{x})} \right]$$

$$= - \mathcal{L}(\theta, \phi|x, \hat{x}) + D_{KL}(q_\phi(z|x)\|p_\theta(z,\hat{x}))$$

$$\geq - \mathcal{L}(\theta, \phi|x, \hat{x}). \quad (20)$$

In the formula, the term $- \mathcal{L}(\theta, \phi|x, \hat{x})$ is referred to as the evidence lower bound (ELBO), or the variational lower bound. It can be expressed as:

$$- \mathcal{L}(\theta, \phi|x, \hat{x})$$

$$= \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \frac{p_\theta(x,z,\hat{x})}{q_\phi(z|x)} \right]$$

$$= \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \frac{p_\theta(x,z,\hat{x})}{q_\phi(z|x)} \right] + \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \frac{q_\phi(z|x)}{p_\theta(x,z,\hat{x})} \right]$$

$$= \mathbb{E}_{z \sim q_\phi(z|x)} \left[ \log \frac{p_\theta(x,z,\hat{x})}{q_\phi(z|x)} \right] + D_{KL}(q_\phi(z|x)\|p_\theta(x,z,\hat{x})) \quad (21)$$

Consequently, the loss function can be formulated as follows:

$$\mathcal{L}(\theta, \phi) = \mathbb{E}_{x \sim D'} \mathbb{E}_{\hat{x} \sim \hat{p}(\hat{x}|x)} \left[ - \mathcal{L}(\theta, \phi|x, \hat{x}) \right], \quad (22)$$

where

$$\mathcal{L}(\theta, \phi|x, \hat{x})$$

$$= \mathbb{E}_{z \sim q_\phi(z|x)} \left[ - \log p_\theta(x,z,\hat{x}) \right] + D_{KL}(q_\phi(z|x)\|p_\theta(z,\hat{x})). \quad (23)$$

In the formula, the first term $\mathcal{L}_{rec}$ is referred to as the reconstruction loss, as it urges the decoder to reconstruct the input data $x$. The second term $\mathcal{L}_{prior}$ is referred to as the prior loss, as it regularizes the posterior distribution $q_\phi(z|x)$ of the latent variable to approximate the prior distribution $p_\theta(z)$. In practice, the prior distribution $p_\theta(z)$ is commonly taken as $\mathcal{N}(0, I)$, and a hyperparameter is often introduced as the coefficient for the prior loss. Consequently, the loss function can be expressed as:

$$\mathcal{L}(\theta, \phi) = \mathbb{E}_{x \sim D'} \mathbb{E}_{\hat{x} \sim \hat{p}(\hat{x}|x)} \left[ - \mathcal{L}(\theta, \phi|x, \hat{x}) \right], \quad (24)$$

where

$$\mathcal{L}(\theta, \phi|x, \hat{x}) = \mathcal{L}_{rec}(\theta, \phi|x, \hat{x}) + \beta \cdot \mathcal{L}_{prior}(\phi|x),$$

$$\mathcal{L}_{rec}(\theta, \phi|x, \hat{x}) = \mathbb{E}_{x \sim q_\phi(z|x)} \left[ - \log p_\theta(x,z,\hat{x}) \right],$$

$$\mathcal{L}_{prior}(\phi|x) = D_{KL}(q_\phi(z|x)\|\mathcal{N}(0, I)). \quad (25)$$

In G2MILP, the loss function is instantiated as:

$$\mathcal{L}(\theta, \phi) = \mathbb{E}_{G \sim D} \mathbb{E}_{\hat{G} \sim \hat{p}(\hat{G}|G)} \left[ - \mathcal{L}(\theta, \phi|G, \hat{G}) \right], \quad (26)$$

where

$$\mathcal{L}(\theta, \phi|G, \hat{G}) = \mathcal{L}_{rec}(\theta, \phi|G, \hat{G}) + \beta \cdot \mathcal{L}_{prior}(\phi|G),$$

$$\mathcal{L}_{rec}(\theta, \phi|G, \hat{G}) = \mathbb{E}_{Z \sim q_{\phi}(Z|G)} \left[ - \log p_\theta(G|Z, \hat{G}) \right],$$

$$\mathcal{L}_{prior}(\phi|G) = D_{KL}(q_\phi(Z|G)\|\mathcal{N}(0, I)). \quad (27)$$

A.2 G2MILP Implementation

A.2.1 Encoder

The encoder implements $q_\phi(Z|G)$ in Equation 27. Given a bipartite graph $G = (V \cup W, E)$ equipped with the feature metrics $(V, W, E)$, we employ a GNN structure with parameters $\phi$ to extract the representations. Specifically, we utilize MLPs as embedding layers to obtain the initial embeddings $h_v^{(0)}, h_w^{(0)}$, and $h_{eij}$, given by:

$$h_v^{(0)} = MLP_\phi(v_i), \quad h_w^{(0)} = MLP_\phi(w_j), \quad h_{eij} = MLP_\phi(e_{ij}). \quad (28)$$

Next, we perform $K$ graph convolution layers, with each layer in the form of two interleaved half-convolutions. The convolution layer is defined as follows:

$$h_v^{(k+1)} = MLP_\phi \left( h_v^{(k)}, \sum_{j:e_{ij} \in E} MLP_\phi \left( h_v^{(k)}, h_{eij}, h_{ij}^{(k)} \right) \right),$$

$$h_w^{(k+1)} = MLP_\phi \left( h_w^{(k)}, \sum_{i:e_{ij} \in E} MLP_\phi \left( h_v^{(k)}, h_{eij}, h_{w}^{(k)} \right) \right). \quad (29)$$

The convolution layer is followed by two GraphNorm layers, one for constraint vertices and the other for variable vertices. We employ a concatenation Jumping Knowledge layer to aggregate information from all $K$ layers and obtain the node representations:

$$h_v = MLP_\phi \left( CONCAT \left( h_v^{(k)} \right) \right),$$

$$h_w = MLP_\phi \left( CONCAT \left( h_w^{(k)} \right) \right). \quad (30)$$

The obtained representations contain information about the instances. Subsequently, we use two MLPs to output the mean and log variance, and then sample the latent vectors for each vertex from a Gaussian distribution as follows:

$$z_v \sim \mathcal{N} \left( MLP_\phi \left( h_v \right), \exp MLP_\phi \left( h_v \right) \right),$$

$$z_w \sim \mathcal{N} \left( MLP_\phi \left( h_w \right), \exp MLP_\phi \left( h_w \right) \right).$$
A.2.2 Decoder

The decoder implements $p_{\theta}(G|Z, \hat{G})$ in Equation 27. It utilizes a GNN structure to obtain the representations, which has the same structure as the encoder GNN, but is with parameters $\theta$ instead of $\phi$. To encode the masked graph, we assign a special $\text{[mask]}$ token to the masked vertex $\hat{v}$. Its initial embedding $\hat{h}_{\text{[mask]}}^{(0)}$ is initialized as a special embedding $h_{\text{[mask]}}$. We mask all edges between $\hat{v}$ and the variable vertices and add virtual edges. In each convolution layer, we apply a special update rule for $\hat{v}$:

$$h_{\hat{v}}^{(k+1)} = \text{MLP}_\theta \left( h_{\hat{v}}^{(k)}, \text{MEAN}_{w_j \in W} \left( h_{w_j}^{(k+1)} \right) \right),$$

$$h_{w_j}^{(k+1)} = \text{MLP}_\theta \left( h_{w_j}^{(k+1)}, h_{\hat{v}}^{(k+1)} \right).$$

This updating is performed after each convolution layer, allowing $\hat{v}$ to aggregate and propagate the information from the entire graph.

The obtained representations are used for the four networks—Bias Predictor, Degree Predictor, Logits Predictor, and Weights Predictor—to determine the generated graph. The details of these networks have been described in the main paper. Here we provide the losses for the four prediction tasks. In the following context, the node features, e.g., $h_{v, \tilde{v}}$, refer to those from $\tilde{G}$ obtained by the decoder GNN.

1. **Bias Prediction Loss:**
   $$L_1(\theta, \phi|G, \tilde{G}) = \text{MSE} \left( \sigma \left( \text{MLP}_{\text{bias}}([h_v, z_v]) \right), b^*_v \right). \quad (31)$$

2. **Degree Prediction Loss:**
   $$L_2(\theta, \phi|G, \tilde{G}) = \text{MSE} \left( \sigma \left( \text{MLP}_{\text{deg}}([h_v, z_v]) \right), d^*_v \right). \quad (32)$$

3. **Logits Prediction Loss:**
   $$L_3(\theta, \phi|G, \tilde{G}) = -E_{(v,u) \sim \text{pos}} \left[ \log(\hat{\delta}_{v,u}^*) \right] - E_{(v,u) \sim \text{neg}} \left[ \log(1 - \hat{\delta}_{v,u}^*) \right], \quad (33)$$
   where
   $$\hat{\delta}_{v,u}^* = \sigma \left( \text{MLP}_{\text{logits}}([h_u, z_u]) \right). \quad (34)$$

4. **Weights Prediction Loss:**
   $$L_4(\theta, \phi|G, \tilde{G}) = \text{MSE} \left( \sigma \left( \text{MLP}_{\text{weight}}([h_u, z_u]) \right), e^*_v \right). \quad (35)$$

With these four prediction tasks, the reconstruction loss in Equation 27 is instantiated as:

$$L_{\text{rec}}(\theta, \phi|G, \tilde{G}) = \sum_{i=1}^{4} \alpha_i \cdot L_i(\theta, \phi|G, \tilde{G}). \quad (36)$$

### Appendix B

**Experimental Details**

#### B.1 Dataset

The three commonly used datasets, namely MIS, SetCover, and MIK, are the same as those used in [12]. Nurse Scheduling contains a group of 4 instances from MIPLIB 2017: nursesched-medium04 and nursesched-sprint-hidden09 for training, and nursesched-sprint02 and nursesched-sprint-late03 for test. We summarize some statistics of these datasets in Table 11.

### Table 11

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MIS</th>
<th>SetCover</th>
<th>MIK</th>
<th>Nurse Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>1000</td>
<td>1000</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>V</td>
<td>$</td>
<td>1953</td>
<td>500</td>
</tr>
<tr>
<td>$</td>
<td>W</td>
<td>$</td>
<td>500</td>
<td>1000</td>
</tr>
</tbody>
</table>

#### B.2 Hyperparameters

We report some important hyperparameters in this section. Further details can be found in our code once the paper is accepted to be published.

We run our model on a single GeForce RTX 3090 GPU. The hidden dimension and the embedding dimension are set to 16. The depth of the GNNs is 6. Each MLP has one hidden layer and uses $\text{ReLU}(\cdot)$ as the activation function.

In this work, we simply set all $\alpha_i$ to 1. We find that the choice of $\beta$ significantly impacts the model performance. For MIS, we set $\beta$ to 0.00045. For SetCover, MIK and Nurse Scheduling, we apply a sigmoid schedule [46] to let $\beta$ to reach 0.0005, 0.001, and 0.001, respectively. We employ the Adam optimizer, train the model for 20,000 steps, and choose the best checkpoint based on the average error in solving time and the number of branching nodes. The learning rate is initialized to 0.001 and decays exponentially. For MIS, SetCover, and MIK, we set the batch size to 30. Specifically, to provide more challenging prediction tasks in each batch, we sample 15 graphs and use each graph to derive 2 masked ones for training. For Nurse Scheduling, we set the batch size as 1 due to the large size of each graph. For fine-tuning in hard MILP generation, we run 2000 steps in each iteration, with a small learning rate of 0.0001, and we generate 200 steps each 150 steps to identify the best checkpoints.

### Appendix C

**Additional Results**

#### C.1 Comparison with G2SAT

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MIS</th>
<th>SetCover</th>
<th>MIK</th>
<th>Nurse Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>V</td>
<td>$</td>
<td>1953</td>
<td>500</td>
</tr>
<tr>
<td>$</td>
<td>W</td>
<td>$</td>
<td>500</td>
<td>1000</td>
</tr>
</tbody>
</table>

Results of G2SAT on MIS. In the table, “sim” denotes similarity score (higher is better), “time” denotes solving time, and “#branch” denotes the number of branching nodes, respectively. Numbers in brackets denote relative errors (lower is better).

<table>
<thead>
<tr>
<th>Simulation</th>
<th>sim</th>
<th>time (s)</th>
<th>#branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Set</td>
<td>0.998</td>
<td>0.349</td>
<td>16.09</td>
</tr>
<tr>
<td>G2SAT</td>
<td>0.572</td>
<td>0.014</td>
<td>2.11</td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.01$)</td>
<td>0.997</td>
<td>0.354</td>
<td>15.03</td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.1$)</td>
<td>0.895</td>
<td>0.214</td>
<td>4.61</td>
</tr>
</tbody>
</table>

We conduct an additional experiment that transfers G2SAT to a special MILP dataset, MIS, in which all coefficients are 1.0 and thus the instances can be modeled as homogeneous bipartite graphs. We apply G2SAT to learn to generate new graphs and convert them to MILPs. Results are in Table 12. The results show that G2MILP significantly outperforms G2SAT on the special cases.
C.2 Masking Process

C.2.1 Masking Variables.

In the mainbody, for simplicity, we define the masking process of uniformly sampling a constraint vertex $\tilde{v} \sim \mathcal{U}(\mathcal{V})$ to mask, while keeping the variable vertices unchanged. We implement different versions of G2MILP that allow masking and modifying either constraints, variables, or both. The results are in Table 13.

<table>
<thead>
<tr>
<th>Model</th>
<th>sim</th>
<th>time (s)</th>
<th>#branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>G2MILP ($\eta = 0.01$)</td>
<td>0.997</td>
<td>0.198</td>
<td>175.35</td>
</tr>
<tr>
<td>v</td>
<td>0.183 (7.5%)</td>
<td>136.68 (22.0%)</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0.169 (17.1%)</td>
<td>167.44 (4.5%)</td>
<td></td>
</tr>
<tr>
<td>vc</td>
<td>0.186 (6.1%)</td>
<td>155.40 (11.4%)</td>
<td></td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.05$)</td>
<td>0.989</td>
<td>0.169 (17.1%)</td>
<td>167.44 (4.5%)</td>
</tr>
<tr>
<td>v</td>
<td>0.176 (11.1%)</td>
<td>136.68 (22.0%)</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0.148 (25.3%)</td>
<td>150.90 (13.9%)</td>
<td></td>
</tr>
<tr>
<td>vc</td>
<td>0.147 (25.3%)</td>
<td>142.70 (18.6%)</td>
<td></td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.1$)</td>
<td>0.964</td>
<td>0.117 (40.9%)</td>
<td>169.63 (3.3%)</td>
</tr>
<tr>
<td>v</td>
<td>0.172 (13.1%)</td>
<td>136.67 (22.1%)</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0.148 (25.3%)</td>
<td>142.70 (18.6%)</td>
<td></td>
</tr>
<tr>
<td>vc</td>
<td>0.115 (41.9%)</td>
<td>112.29 (35.9%)</td>
<td></td>
</tr>
</tbody>
</table>

C.2.2 Orders of Masked Constraints

We also investigate different orders of masking constraint vertices, including uniformly sampling and sampling according to the vertex indices. Results are in Table 14. We find that uniformly sampling achieves the best performance. Sampling according to indices leads to a performance decrease, maybe because near constraints are relevant and lead to error accumulation. We think these results are interesting, and will study it in the future work.

<table>
<thead>
<tr>
<th>Model</th>
<th>sim</th>
<th>time (s)</th>
<th>#branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uni</td>
<td>0.953</td>
<td>0.129 (35.1%)</td>
<td>235.35 (34.2%)</td>
</tr>
<tr>
<td>Random</td>
<td>0.840</td>
<td>0.004 (97.9%)</td>
<td>0.00 (100%)</td>
</tr>
<tr>
<td>Idx↗</td>
<td>0.892</td>
<td>0.054 (72.7%)</td>
<td>108.30 (38.2%)</td>
</tr>
<tr>
<td>Random</td>
<td>0.773</td>
<td>0.002 (98.9%)</td>
<td>0.00 (100%)</td>
</tr>
<tr>
<td>Idx↘</td>
<td>0.925</td>
<td>0.027 (86.2%)</td>
<td>31.53 (82.0%)</td>
</tr>
<tr>
<td>Random</td>
<td>0.827</td>
<td>0.003 (98.6%)</td>
<td>0.00 (100%)</td>
</tr>
</tbody>
</table>

C.3 Structural Distributional Similarity

In the mainbody, for a fair comparison, we exclude statistics that remain constant in our approach, such as problem size and objective coefficients. However, these statistics are also important features for MILPs. In this section, we incorporate three additional statistics in the computing of similarity scores: (1) mean of objective coefficients $\mu_c$, (2) std of objective coefficients $\sigma_c$, and (3) the ratio of continuous variables. With these additional metrics, we recompute the structural similarity scores and updated the results in both Table 4 and Table 13. The new results are in Table 15 and Table 16, respectively. From the results, we can still conclude that G2MILP outperforms all baselines, further supporting the effectiveness of our proposed method.

<table>
<thead>
<tr>
<th>Model</th>
<th>sim</th>
<th>time (s)</th>
<th>#branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bowly</td>
<td>0.144</td>
<td>0.150</td>
<td>-</td>
</tr>
<tr>
<td>Random</td>
<td>0.722</td>
<td>0.791</td>
<td>0.971</td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.01$)</td>
<td>0.997</td>
<td>0.874</td>
<td>0.994</td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.05$)</td>
<td>0.951</td>
<td>0.833</td>
<td>0.969</td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.1$)</td>
<td>0.921</td>
<td>0.834</td>
<td>0.930</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>v</th>
<th>c</th>
<th>cv</th>
</tr>
</thead>
<tbody>
<tr>
<td>G2MILP ($\eta = 0.01$)</td>
<td>0.998</td>
<td>0.988</td>
<td>0.985</td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.05$)</td>
<td>0.996</td>
<td>0.968</td>
<td>0.967</td>
</tr>
<tr>
<td>G2MILP ($\eta = 0.1$)</td>
<td>0.996</td>
<td>0.928</td>
<td>0.912</td>
</tr>
</tbody>
</table>