ACM-MILP: Adaptive Constraint Modification via Grouping and Selection for Hardness-Preserving MILP Instance Generation

Ziao Guo [1] **Yang Li** [1] **Chang Liu** [1] **Wenli Ouyang** [2] **Junchi Yan** [1]

Abstract

Data plays a pivotal role in the development of both classic and learning-based methods for Mixed-Integer Linear Programming (MILP). However, the scarcity of data in real-world applications underscores the necessity for MILP instance generation methods. Currently, these methods primarily rely on iterating random single-constraint modifications, disregarding the underlying problem structure with constraint interrelations, thereby leading to compromised quality and solvability. In this paper, we propose ACM-MILP, a framework for MILP instance generation, to achieve adaptive constraint modification and constraint interrelation modeling. It employs an adaptive constraint selection mechanism based on probability estimation within the latent space to preserve instance characteristics. Meanwhile, it detects and groups strongly related constraints through community detection, enabling collective modifications that account for constraint dependencies. Experimental results show significant improvements in problem-solving hardness similarity under our framework. Additionally, in the downstream task, we showcase the efficacy of our generated instances for hyperparameter tuning. Source code is available: https://github.com/Thinklab-SJTU/ACM-MILP.

1. Introduction

Mixed-integer linear programming (MILP), as a significant Combinatorial Optimization (CO) problem, is of central importance in operation research and computer science and serves broad applications in various real-world scenarios including planning (da Silva et al., 2006), scheduling (Muckstadt & Wilson, 1968), routing (Dong et al., 2014), etc. Beyond traditional branch-and-bound solvers (Gurobi Optimization, LLC, 2023; Cplex, 2009; Bestuzheva et al., 2021a;b), recent trends have led to a proliferation of machine learning (ML) approaches in a data-driven paradigm, generally assisting traditional solvers in solving process (Paulus et al., 2022; Zhang et al., 2023b; Li et al., 2023b; 2024; Zhang et al., 2023a). These approaches hold promise for enhancing both speed and quality in problem-solving.

In particular, the pivotal role of high-quality instances has become increasingly prominent. The process of testing and tuning solver hyperparameters (Hutter et al., 2010), which is crucial for solver efficiency and accuracy, often heavily relies on comprehensive datasets that faithfully reflect the diversity and complexity of real-world scenarios (Li et al., 2023a). Meanwhile, the effectiveness of data-driven ML approaches naturally depends on the availability of extensive and representative data (Bengio et al., 2021). However, due to the complexity and specificity of MILP, as well as data privacy and proprietary constraints (Roling et al., 2008; Freund & Naor, 2004; Sakuma & Kobayashi, 2007), available real-world data are rather scarce (Gleixner et al., 2021), which poses obstacles to both classic and ML-based MILP solvers and heightens the need for efficient instance generation.

Efforts have been made to address this data bottleneck, primarily stemming from two lines toward instance generation. Hand-crafted instance generation methods either leverage specialized strategies to generate instances of specific problems that can be readily reduced to MILP (Bergman et al., 2016; Balas & Ho, 1980; Leyton-Brown et al., 2000), or generate new instances by sampling in an encoding space while controlling a few specific statistics (Smith-Miles & Bowly, 2015; Bowly, 2019). These methods heavily rely on expert knowledge and manual efforts, failing to adaptively unravel specific data characteristics and maintain general applicability. In comparison, learning-based generation methods leveraging deep neural networks can naturally accommodate the capture of global instance features through data-driven training. This line of work is still in its early stage, which primarily encompasses G2MILP (Geng et al., 2023), the first learning-based MILP generation model with a masked

[1]School of Artificial Intelligence & Department of Computer Science and Engineering & MoE Lab of AI, Shanghai Jiao Tong University, Shanghai, China [2]AI Lab, Lenovo Research, Beijing, China. Correspondence to: Junchi Yan <yanjunchi@sjtu.edu.cn>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

Variational Autoencoder (VAE) paradigm. It represents a MILP instance as a bipartite graph and iteratively corrupts and replaces a constraint node using sampled latent vectors. However, the iterating of *random single-constraint* modification overlooks the inherent problem structure with constraint correlations. Specifically, it randomly selects the constraints during modification, which may inadvertently compromise crucial constraints and change the nature of the problem, leading to quality degradation of generated instances. Additionally, the strategy of modifying only a single constraint at a time can potentially disrupt the intrinsic interrelations among constraints and even significantly alter the feasible region. Such disruptions may undermine the coherence and the quality of the generated instances.

In this paper, we address two critical questions for constraint modeling in the context of MILP instance generation: **Q1.** *Which modifications of constraints in a specific instance distribution can best preserve the inherent properties of the instances?* **Q2.** *How to recognize and process the interrelations among constraints?* Based on these two aspects, we propose a learning-based generative framework ACM-MILP for MILP instance generation, achieving adaptive and correlation-aware constraint reconstruction via constraint grouping and probability modeling of substructures. Specifically, we transform MILP instances into bipartite graph representations, where constraints and variables are vertices and non-zero constraint coefficients are edges. We encode the vertices in the bipartite graphs utilizing the principles of VAE (Kipf & Welling, 2016; Kingma & Welling, 2013), which naturally computes and retains the sampling probabilities of constraints, distinguishing them into generic and instance-unique constraints. We selectively reconstruct these unique constraints to maintain the structural characteristics of the data distribution in an iterative way. Furthermore, we identify the tightly coupled constraints via community detection and simultaneously modify a community of constraints to maintain their interrelatedness.

Experiments are conducted to evaluate the quality of generated instances in two folds. First, we measure the statistical distributional similarity between generated and raw training instances by multiple graph statistics. Next, we solve the instances using the popular MILP solver Gurobi (Gurobi Optimization, LLC, 2023) to measure the computational hardness similarity via the solving time and the number of branching nodes (Li et al., 2023a; Balyo et al., 2020). Empirical results demonstrate the superiority of ACM-MILP in resembling computational hardness compared to previous state-of-the-art while preserving statistical distributional similarity. Moreover, to show the applications on the downstream tasks, we examine the correlation in solving time with different solver hyperparameter configurations between training and generated instances and tune the hyperparameters of Gurobi utilizing the generated instances. Our method shows promising potential and effectiveness in hyperparameter tuning on MILP solvers.

The main contributions of the paper are as follows.

1) We propose ACM-MILP, a framework for MILP instance generation through adaptive constraint modification. This framework includes constraint grouping to maintain the intrinsic interrelations among constraints and constraint selection to preserve the inherent problem structures.

2) We conduct MILP instance generation experiments on 3 datasets, verifying the significant effectiveness of ACM-MILP in generating hardness-preserving MILP instances, and provide ablation studies alongside further analysis.

3) We demonstrate the promising potential of ACM-MILP in the downstream task of solver hyperparameter tuning.

2. Related Work

Beyond generative models on image and text data (Li et al., 2022; 2023c), there are recent works on instance generation for more complex combinatorial problems, ranging from satisfiability problems (You et al., 2019; Li et al., 2023a; Chen et al., 2024) to LP (Bowly et al., 2020). Here we mainly discuss those on MILP.

Heuristic MILP Generation. Researches on heuristic MILP generation are primarily divided into two categories. The first includes various problem-specific generation methods (Cornuéjols et al., 1991; Drugan, 2013; Drexl et al., 2000). These approaches focus on leveraging mathematical formulations and expert knowledge to generate specific instances such as TSP (Vander Wiel & Sahinidis, 1995) or knapsack (Atamtürk, 2003). These methods are not generalizable across problem domains. The second aims to generate general MILP instances. Bowly (2019) generated new instances by sampling in an encoding space while manually controlling some graph statistics. They achieved only partial matches to structural metrics through manual adjustments, lacking the ability to fully encapsulate the nuanced and dynamic nature of real-world data distributions. ACM-MILP generates general MILP instances by adaptively learning the rich features of instances in a data-driven manner.

Learning-based MILP Generation. In Geng et al. (2023), a learning-based MILP generative model is proposed with a masked VAE paradigm, which iteratively reconstructs a constraint. All constraints are uniformly modified, which can be detrimental to preserving the inherent properties of instances. Additionally, their modification approach may disrupt the interrelations among the constraints. ACM-MILP selectively modifies constraints to preserve instance properties and handles the interrelations among the constraints.

In this paper, we follow the test setting of existing related

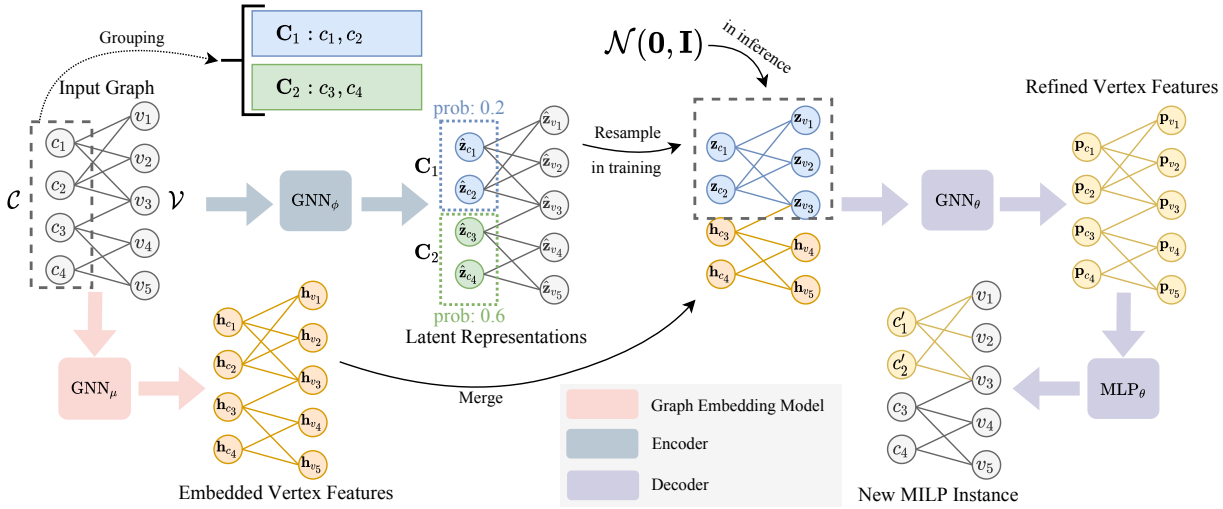


Figure 1. Overview of our ACM-MILP. The input instance is represented as a bipartite graph. $\mathcal{C}$ denotes the constraint vertex set, and $\mathcal{V}$ denotes the variable vertex set. The constraints are first grouped and divided into communities. The encoder yields the latent representations of the vertices. We compute the Gaussian probabilities for all communities and then select the community with the lowest Gaussian probability for modification. In line with VAE, we resample the representations in training while directly sampling from a standard Gaussian in inference. The representations are merged with the embedded vertex features yielded by the graph embedding model. The decoder generates new constraint vertices in the selected community, as well as the edges connected to the community.

works that use the generated instance to better fine-tune the solvers' hyperparameters or independently evaluate the hardness and similarity to the original instances.

3. Methodology

In this section, we present the insights and details of our ACM-MILP framework. We start by introducing the preliminary background on MILP instance representation in Sec. 3.1. Then, we concentrate on answering the two questions raised in Sec. 1 and elaborate on the constraint grouping and adaptive selection in Sec. 3.2. Finally, we introduce the overall pipeline and model implementation, as well as some important training details in Sec. 3.3. Fig. 1 shows the overall procedure of our ACM-MILP framework.

3.1. Preliminary

Given a set of variables $\mathbf{x} \in \mathbb{R}^n$, the MILP problem is formulated as follows:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \tilde{\mathbf{c}}^\top \mathbf{x}, \quad \textbf{s.t. } \mathbf{A}\mathbf{x} \leq \mathbf{b}, \; x_i \in \mathbb{Z}, \; \forall i \in \mathcal{I}, \tag{1}$$

where $\tilde{\mathbf{c}} \in \mathbb{R}^n$ is the objective coefficient, $\mathbf{A} \in \mathbb{R}^{m\times n}$ is the constraint coefficient matrix, and $\mathbf{b} \in \mathbb{R}^m$ is the bias. $\mathcal{I}$ is the index set indicating the integrality of the variables.

In line with the literature (Zhang et al., 2023b), we represent each MILP instance with a weighted bipartite graph (Gasse et al., 2019; Nair et al., 2021) $\mathcal{G} = (\mathcal{C}, \mathcal{V}, \mathcal{E})$. $\mathcal{C}$ and $\mathcal{V}$ respectively represent the two parts of $\mathcal{G}$, while $\mathcal{E}$ denotes the edge set of $\mathcal{G}$. Specifically, $\mathcal{C} = \{c_1, c_2, \cdots, c_m\}$ contains m constraint vertices, where vertex c_i corresponds to the i-th constraint. The feature for vertex c_i, denoted as $\mathbf{c}_i$, is a 2-dimensional vector including a bias term b_i and a positional encoding (Vaswani et al., 2017) $\frac{i}{m}$. b_i is the i-th element of $\mathbf{b}$. The positional encoding is utilized to enhance representation (Chen et al., 2023). $\mathcal{V} = \{v_1, v_2, \cdots, v_n\}$ contains n variable vertices, where vertex v_i corresponds to the i-th variable. The variable feature $\mathbf{v}_i$ is a 9-dimensional vector calculated by the combinatorial optimization environment Ecole (Prouvost et al., 2020), including the objective coefficient, the variable type, and the variable bounds. $\mathcal{E} = \{e_{ij} | c_i \in \mathcal{C}, v_j \in \mathcal{V}\}$ contains edges in $\mathcal{G}$. e_{ij} connects constraint vertex c_i and variable vertex v_j with edge weight A_{ij}. A_{ij} is the i-th row and j-th column element of constraint coefficient matrix $\mathbf{A}$. $A_{ij} = 0$ denotes no edge exists. The feature $\mathbf{e}_{ij}$ for edge e_{ij} is the edge weight A_{ij}. Please refer to Appendix A for more details on the graph features.

3.2. Constraint Grouping and Adaptive Selection

3.2.1. Raw Constraint Grouping

We first illustrate the significance of processing the interrelations among constraints. Constraints are crucial in MILP, as they determine the feasible region for variables, essentially defining the solver's search space. The similarity in solving the original and generated instances is directly linked to how closely their constraint-based feasible regions match.

Moreover, the feasible region is not determined by a single constraint but by the joint combination of multiple con-

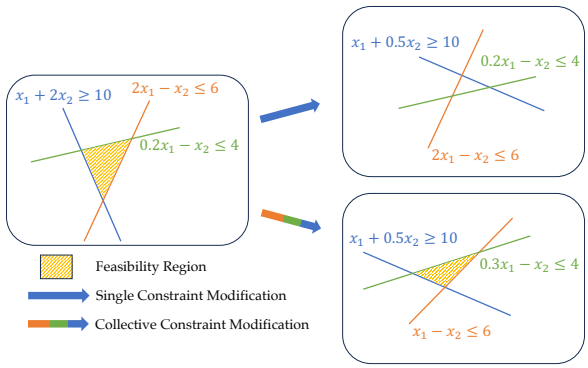


Figure 2. An example of single and collective constraint modification. There are three strongly related constraints on x_1 and x_2, and they jointly form the feasible region. After modifying a single constraint, the feasible region becomes empty. The collective constraint modification means modifying the three constraints simultaneously and results in a similar feasible region.

straints. Modifying just one constraint while keeping others unchanged can significantly alter the feasible region, potentially rendering the problem unsolvable. Therefore, one constraint should be modified in conjunction with other closely related constraints to maintain the similarity in the feasible region. Fig. 2 illustrates an example. The single-constraint modification leads to an empty feasible region, while modifying all related constraints results in a similar feasible region to the original. It indicates the necessity and significance of operating strongly related constraints together by grouping.

Naturally, the next issue is to find strongly related constraints for grouping. Intuitively, the constraints connected with similar or common variables are strongly related and should be grouped or coupled. Therefore, we resort to community detection for constraint grouping. Without loss of generality, here we adopt the Louvain (Blondel et al., 2008) algorithm, a heuristic method based on modularity (Clauset et al., 2004) to extract the community structure of a graph. Here the modularity Q is defined as:

$$Q = \frac{1}{2\tau} \sum_{ij} \left(G_{ij} - \gamma \frac{k_i k_j}{2\tau} \right) \delta(C_i, C_j), \tag{2}$$

where τ is the number of edges, $\mathbf{G}$ is the **unweighted** adjacency matrix of the bipartite graph $\mathcal{G}$, G_{ij} is the i-th row and j-th column element of $\mathbf{G}$ and k_i is the degree of the i-th vertex. C_i is the index of the community that contains the i-th vertex. δ is an indicator, where $\delta(C_i, C_j) = 1$ if $C_i = C_j$ else 0. γ is a hyperparameter. We set $\mathbf{G}$ to the unweighted adjacency matrix because the edge weight of $\mathcal{G}$ is the constraint coefficient and cannot represent relevance.

The Louvain algorithm first assigns each vertex to its own community and moves each vertex to its neighbors' community to find the maximum positive modularity gain. It stops when no modularity gain is achieved. In this way, we can group the constraint vertices connected to similar variables into the same community. Since our focus is solely on grouping the constraints, we can further refine the results provided by the Louvain algorithm by removing variable vertices and retaining only the constraint vertices. Please refer to Appendix B.1 for more details.

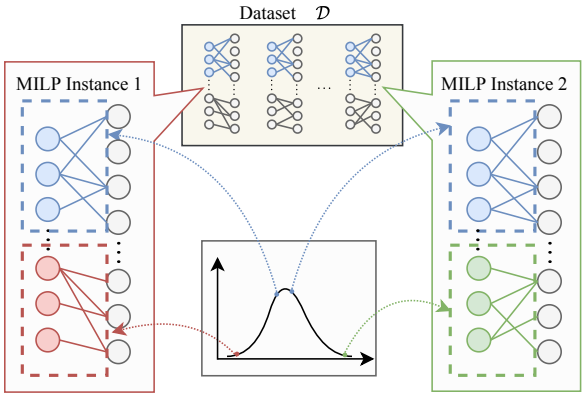


Figure 3. Our insight on constraint group selection. The constraint groups with high sampling probabilities (blue vertices) are more generic in the dataset, while those with low sampling probabilities (red and green vertices) are more instance-specific.

3.2.2. Constraint Group Selection

The selection of constraints has a significant impact on the quality of generated instances. Recall that in Geng et al. (2023), the modified constraints are randomly selected. However, this approach might lead to the modification of numerous generic constraints, which breaks the specific problem property related to computational hardness. We aim to modify the constraints by which we can best preserve the inherent properties of the instances.

To address the issue, we first take a batch of MILP instances that satisfy the same distribution. Utilizing VAE, we encode the constraints of all instances into a latent space and guide them to conform to a standard Gaussian distribution. By VAE, the sampling probabilities of constraints are naturally preserved and can be used to differentiate the constraints. A higher sampling probability of a constraint corresponds to more occurrences in the distribution, thus indicating that the constraint is more generic and contains inherent properties. In contrast, those with lower sampling probabilities are more instance-specific. Fig. 3 illustrates our insight.

When computing the sampling probability, we take the characteristics of instances into consideration by moving the center of the Gaussian distribution to the mean of the constraint representations for each instance. Therefore, the sampling probability $P_j(c)$ of a constraint c in the j-th instance can be calculated by the probability density function of Gaussian distribution:

$$P_j(c) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{||\hat{\mathbf{z}}_c - \hat{\mathbf{z}}_j||^2}{2}\right), \tag{3}$$

where $\hat{\mathbf{z}}_c$ is the Gaussian center of constraint vertex c given by the VAE encoder. $\hat{\mathbf{z}}_j$ is the mean of the constraint representations in instance j which can be calculated via $\hat{\mathbf{z}}_j = \frac{1}{|\mathcal{C}_j|}\sum_{c\in\mathcal{C}_j}\hat{\mathbf{z}}_c$. $\mathcal{C}_j$ is the constraint set in instance j.

Furthermore, as described in Sec. 3.2.1, the constraints are grouped as communities. We select and modify a whole community in practice. Therefore, we measure the sampling probability of community $\mathbf{C}$ with the average constraint sampling probability in community $\mathbf{C}$, denoted as $P_j(\mathbf{C})$:

$$P_j(\mathbf{C}) = \frac{1}{|\mathbf{C}|}\sum_{c\in\mathbf{C}} P_j(c), \tag{4}$$

where $|\mathbf{C}|$ denotes the number of constraint vertices in $\mathbf{C}$. The sampling probability $\hat{P}_j(\mathbf{C})$ of community $\mathbf{C}$ is computed by adding a normalization on $P_j(\mathbf{C})$ through:

$$\hat{P}_j(\mathbf{C}) = \frac{P_j(\mathbf{C})}{\sum_{\mathbf{C}'\in\mathbb{C}} P_j(\mathbf{C}')}, \tag{5}$$

where $\mathbb{C}$ is the set containing all constraint communities. To preserve the inherent properties of the MILP instances, we select the instance-specific communities, i.e., those with lower sampling probabilities for modification.

3.3. Generation Pipeline and Implementation

3.3.1. Generation Paradigm

Inspired by Geng et al. (2023), we devise a VAE paradigm for MILP instance generation. Please note that the instances have been transformed into bipartite graphs. The instance generation is converted into a graph generation task.

Given a MILP instance and its corresponding bipartite graph $\mathcal{G}$ drawn from dataset $\mathcal{D}$, we aim to generate a community of constraint vertices to replace the original community in $\mathcal{G}$, and thus generate a new graph $\hat{\mathcal{G}}$. Following the standard VAE framework (Kingma & Welling, 2013; Kipf & Welling, 2016), we introduce an encoder or recognition model with parameter $\boldsymbol{\phi}$, and a decoder or generative model with parameter $\boldsymbol{\theta}$. $\mathbf{C}$ is the community to be replaced (assume $c_i, \cdots, c_j \in \mathbf{C}$, and $v_k, \cdots, v_l$ are connected variables). $\mathbf{C}$ is selected as described in Sec. 3.2.2 with encoder parameter $\boldsymbol{\phi}$, denoted by $\mathbf{C} \sim \tilde{p}_{\boldsymbol{\phi}}(\mathcal{G})$. We also devise an auxiliary graph embedding model with parameter $\boldsymbol{\mu}$ to provide the features $\mathbf{H}_{\setminus\mathbf{C}}$ of vertices not in or connected with the community $\mathbf{C}$ in decoding, where $\mathbf{H}_{\setminus\mathbf{C}} = (\mathbf{h}_{c_1}, \cdots, \mathbf{h}_{c_{i-1}}, \mathbf{h}_{c_{j+1}}, \cdots, \mathbf{h}_{c_m}, \mathbf{h}_{v_1}, \cdots, \mathbf{h}_{v_{k-1}}, \mathbf{h}_{v_{l+1}}, \cdots, \mathbf{h}_{v_n})$.

Denote $\tilde{\mathcal{G}}$ as the unmodified parts of graph $\mathcal{G}$. In training, we hope to reconstruct $\mathcal{G}$ given $\tilde{\mathcal{G}}$ and $\mathbf{H}_{\setminus\mathbf{C}}$, so we train the decoder by maximizing the log-likelihood $\log p_{\boldsymbol{\theta}}(\mathcal{G}|\tilde{\mathcal{G}}, \mathbf{H}_{\setminus\mathbf{C}}) = \log p_{\boldsymbol{\theta}}(\hat{\mathcal{G}} = \mathcal{G}|\tilde{\mathcal{G}}, \mathbf{H}_{\setminus\mathbf{C}})$. Therefore, training the VAE is to find the best parameters by:

$$\arg\max_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathbb{E}_{\mathcal{G}\sim\mathcal{D}}\mathbb{E}_{\mathbf{C}\sim\tilde{p}_{\boldsymbol{\phi}}(\mathcal{G})} \log p_{\boldsymbol{\theta}}(\mathcal{G}|\tilde{\mathcal{G}}, \mathbf{H}_{\setminus\mathbf{C}}). \tag{6}$$

In line with VAE, we introduce a latent representation $\mathbf{Z}_{\mathbf{C}} = (\mathbf{z}_{c_i}, \cdots, \mathbf{z}_{c_j}, \mathbf{z}_{v_k}, \cdots, \mathbf{z}_{v_l})$ containing latent vectors of constraint vertices in community $\mathbf{C}$ and connected variable vertices. The encoder $q_{\boldsymbol{\phi}}$ learns the distribution parameters $\hat{\mathbf{Z}}_{\mathbf{C}}$ of $\mathbf{Z}_{\mathbf{C}}$. Each vector in $\mathbf{Z}_{\mathbf{C}}$ is sampled from the distribution with parameter $\hat{\mathbf{Z}}_{\mathbf{C}}$ in training. During inference, the latent vectors are independently sampled from a standard Gaussian distribution. The loss is derived by the evidence lower bound (ELBO):

$$\mathcal{L} = \mathbb{E}_{\mathcal{G}\sim\mathcal{D}}\mathbb{E}_{\mathbf{C}\sim\tilde{p}_{\boldsymbol{\phi}}(\mathcal{G})}[\underbrace{\mathbb{E}_{\mathbf{Z}_{\mathbf{C}}\sim q_{\boldsymbol{\phi}}(\mathbf{Z}_{\mathbf{C}}|\mathcal{G})}\left[-\log p_{\boldsymbol{\theta}}(\mathcal{G} \mid \mathbf{Z}_{\mathbf{C}}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus\mathbf{C}})\right]}_{\mathcal{L}_{\mathrm{r}}} + \beta \cdot \underbrace{D_{\mathrm{KL}}\left[q_{\boldsymbol{\phi}}(\mathbf{Z}_{\mathbf{C}} \mid \mathcal{G}) \| p_{\boldsymbol{\theta}}(\mathbf{Z}_{\mathbf{C}})\right]}_{\mathcal{L}_{\mathrm{d}}}], \tag{7}$$

where D_{KL} denotes the KL divergence. $p_{\boldsymbol{\theta}}(\mathbf{Z}_{\mathbf{C}})$ is the prior of $\mathbf{Z}_{\mathbf{C}}$, assumed as Gaussian $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ in consistency with Sec. 3.2.2. $\mathcal{L}_{\mathrm{r}}$ is the reconstruction loss, which quantifies the error in reconstructing the graph $\mathcal{G}$. The divergence loss $\mathcal{L}_{\mathrm{d}}$ encourages the posterior provided by the encoder to conform to a standard Gaussian. This alignment is crucial for matching the sampling distribution used during the inference stage. The hyperparameter β allows for a trade-off between the fidelity of reconstruction and regularization of the latent space.

The overall generation pipeline is concluded as follows:

Constraint Grouping. Given the original MILP instance and its corresponding bipartite graph, the strongly related constraints are coupled through the Louvain algorithm. The detected constraint communities serve as constraint groups.

Constraint Selecting. Utilize the encoder of the VAE to compute the Gaussian centers $\hat{\mathbf{z}}_{c_i}, i = 1, \cdots, n$ of constraint vertices, and calculate the sampling probabilities for constraints by Eq. 3 and then those for communities by Eq. 4 and Eq. 5. The communities are arranged in an ascending order based on their sampling probabilities and are sequentially extracted in the subsequent processes.

Iteratively Replacing. We utilize the VAE paradigm described above to generate a new community of constraints, which is then used to replace the corresponding community in the original instance. This process is continued until the number of replaced constraints $\geq \eta|\mathcal{C}|$, where η is the replacement ratio and $|\mathcal{C}|$ denotes the number of constraints.

Note that we endeavor to make high-quality modifications on constraints, and the variables remain unmodified.

3.3.2. Model Architecture

We elaborate on our model architecture in this section, including the initial embedding layer, encoder, graph embedding model, and decoder.

Initial Embedding Layer. The original features in $\mathcal{G}$ are firstly embedded with an initial embedding layer implemented with MLPs:

$$\mathbf{z}_{c_i}^{(0)} = \text{MLP}^c(\mathbf{c}_i), \mathbf{z}_{v_i}^{(0)} = \text{MLP}^v(\mathbf{v}_i), \mathbf{z}_{e_{ij}}^{(0)} = \text{MLP}^e(\mathbf{e}_{ij}). \tag{8}$$

$\mathbf{z}_{c_i}^{(0)}, \mathbf{z}_{v_i}^{(0)}, \mathbf{z}_{e_{ij}}^{(0)}$ denote the initial embedding of constraint vertex c_i, variable vertex v_i and edge e_{ij}, repectively.

Encoder. The VAE encoder is implemented with a bipartite graph neural network (GNN) combined with two MLPs to obtain the Gaussian distribution parameters $\hat{\mathbf{z}}_u$ and $\hat{\mathbf{z}}_u^{\text{var}}$ of the latent representation:

$$\hat{\mathbf{z}}_u = \text{MLP}_{\boldsymbol{\phi}}^{\text{mean}}\left(\text{GNN}_{\boldsymbol{\phi}}(\mathbf{Z}_c^{(0)}, \mathbf{Z}_e^{(0)}, \mathbf{Z}_v^{(0)})_u\right), \tag{9}$$

$$\hat{\mathbf{z}}_u^{\text{var}} = \text{MLP}_{\boldsymbol{\phi}}^{\text{var}}\left(\text{GNN}_{\boldsymbol{\phi}}(\mathbf{Z}_c^{(0)}, \mathbf{Z}_e^{(0)}, \mathbf{Z}_v^{(0)})_u\right), \tag{10}$$

where $\mathbf{Z}_c^{(0)}, \mathbf{Z}_e^{(0)}, \mathbf{Z}_v^{(0)}$ denote stacks of $\mathbf{z}_c^{(0)}, \mathbf{z}_e^{(0)}$ and $\mathbf{z}_v^{(0)}$, respectively. u is an arbitrary vertex in $\mathcal{G}$ that $u \in \mathcal{C} \cup \mathcal{V}$. $\hat{\mathbf{z}}_u$ and $\hat{\mathbf{z}}_u^{\text{var}}$ serve as the mean and log-variance of the Gaussian, respectively. The latent representation $\mathbf{z}_u$ of vertex u is then sampled in the Gaussian distribution by:

$$\mathbf{z}_u = \mathcal{N}(\hat{\mathbf{z}}_u, \exp(\hat{\mathbf{z}}_u^{\text{var}})), \tag{11}$$

where $\mathbf{z}_u$ is utilized in the decoding phase.

Graph Embedding Model. The function of the graph embedding model is to provide additional information about graph $\mathcal{G}$ for the decoder during the decoding phase, thereby facilitating the decoder in more effectively generating graphs similar to $\mathcal{G}$. We implement it as a bipartite GNN with the same architecture as the encoder GNN:

$$\mathbf{h}_u = \text{GNN}_{\boldsymbol{\mu}}(\mathbf{Z}_c^{(0)}, \mathbf{Z}_e^{(0)}, \mathbf{Z}_v^{(0)})_u, \ u \in \mathcal{C} \cup \mathcal{V}. \tag{12}$$

$\mathbf{h}_u$ is the embedded feature of vertex u in $\mathcal{G}$, and will be utilized in the decoding phase.

Decoder. The VAE decoder aims to reconstruct the selected community $\mathbf{C}$ in $\mathcal{G}$ with $\mathbf{z}_u$ and $\mathbf{h}_u$. It first yields refined vertex features $\tilde{\mathbf{h}}_u$ via merging $\mathbf{z}_u$ and $\mathbf{h}_u$, simply by:

$$\tilde{\mathbf{h}}_u = \begin{cases} \mathbf{z}_u, & \text{if } u \in \mathbf{C} \text{ or connected with } \mathbf{C}, \\ \mathbf{h}_u, & \text{else.} \end{cases} \tag{13}$$

Denote the stack of $\tilde{\mathbf{h}}_u$ as $\tilde{\mathbf{H}}$. Then we apply a GNN to fuse $\tilde{\mathbf{H}}$, and generate the final vertex features $\mathbf{p}_u$:

$$\mathbf{p}_u = \text{GNN}_{\boldsymbol{\theta}}(\tilde{\mathbf{H}}, \mathbf{Z}_e^{(0)})_u, \ u \in \mathcal{C} \cup \mathcal{V}. \tag{14}$$

Denote the stack of $\mathbf{p}_u$ as $\mathbf{P}$. The decoder leverages $\mathbf{P}$ to reconstruct constraints in $\mathbf{C}$.

The reconstruction of a constraint c_i includes constructing the bias b_{c_i} and the constraint coefficient vector $\mathbf{a}_{c_i}$, whereby the constructed constraint is $\mathbf{a}_{c_i}^\top \mathbf{x} \leq b_{c_i}$. However, our objective is to generate a set of constraints in one go, which may require distinct techniques. Regarding biases, we postulate that the correlation of biases between different constraints is not particularly strong, thus enabling us to generate them individually for each constraint. As for the coefficient matrix, which most directly reflects the interconnections and associations between constraints, our approach involves initially predicting a degree for each constraint. Subsequently, for the entire community, we predict a pool of variables that are connected. Within this pool, we select variables connected to each constraint. Finally, we predict the weights of these connections, i.e., the coefficient values. Please refer to Appendix B.3 for more details.

Table 1. Statistical distributional similarity score on datasets.

Replacement Ratio	Model	MIS	SC	CA
/	Bowly	0.319	0.451	0.672
$\eta = 0.05$	Random	0.523	0.643	0.744
	G2MILP	0.974	0.735	0.998
	ACM-MILP	0.756	0.904	0.993
$\eta = 0.1$	Random	0.486	0.558	0.732
	G2MILP	0.947	0.712	0.996
	ACM-MILP	0.668	0.893	0.984
$\eta = 0.2$	Random	0.373	0.504	0.717
	G2MILP	0.823	0.706	0.990
	ACM-MILP	0.613	0.875	0.956

3.3.3. Training Details

We divide the training procedure into two stages as follows.

Stage 1: Communities are randomly selected with the aim of enabling the network to learn the embeddings of all communities. This phase focuses on a broad acquisition of knowledge across various community embeddings.

Stage 2: The communities with low sampling probabilities are chosen for training. The model is then fine-tuned with these specific communities. The objective of this stage is to focus on these least probable communities, which are also the ones typically selected during the inference phase. This targeted approach in the second stage is crucial for refining the network's ability to handle and prioritize communities that are critical for inference.

4. Experiments

4.1. Experimental Settings

We evaluate the similarity between the training and generated instances using graph statistics and computational hardness. Furthermore, we attempt to show that the generated MILP instances can help improve the MILP solvers by tuning their hyperparameters.

Metric: Graph Statistics. In line with Geng et al. (2023), we select 11 classical statistics (Brown et al., 2019) of the

Table 2. Solving time (second) by Gurobi and the relative errors w.r.t. the training sets. We run 5 restarts for each trial and report mean/std.

Replacement Ratio	Model	MIS	SC	CA
0 (Training set)	/	0.315±0.002	1.804±0.013	0.561±0.002
/	Bowly	0.003±0.001	0.012±0.002	0.006±0.001
$\eta = 0.05$	Random	0.509±0.027 (61.6%)	1.705±0.016 (5.5%)	1.062±0.005 (89.3%)
	G2MILP	0.364±0.002 (15.6%)	1.781±0.012 (1.3%)	0.885±0.002 (57.8%)
	ACM-MILP	**0.289±0.002 (8.3%)**	**1.794±0.008 (0.6%)**	**0.523±0.002 (6.8%)**
$\eta = 0.1$	Random	1.184±0.046 (275.9%)	1.542±0.011 (14.5%)	1.402±0.012 (149.9%)
	G2MILP	0.231±0.001 (26.7%)	1.912±0.027 (6.0%)	1.090±0.003 (94.3%)
	ACM-MILP	**0.261±0.005 (17.1%)**	**1.797±0.004 (0.4%)**	**0.498±0.001 (11.2%)**
$\eta = 0.2$	Random	15.016±1.209 (4667.0%)	1.111±0.019 (38.4%)	2.239±0.021 (299.1%)
	G2MILP	0.145±0.002 (54.0%)	1.911±0.022 (5.9%)	1.306±0.005 (132.8%)
	ACM-MILP	**0.234±0.001 (25.7%)**	**1.831±0.014 (1.5%)**	**0.437±0.001 (22.1%)**

Table 3. Number of branching nodes of instances solved by Gurobi and the relative errors w.r.t. the training sets.

Replacement Ratio	Model	MIS	SC	CA
0 (Training set)	/	16.294	839.354	406.107
/	Bowly	0	0	0
$\eta = 0.05$	Random	48.448 (197.3%)	681.406 (18.8%)	844.100 (107.9%)
	G2MILP	23.450 (43.9%)	826.824 (1.5%)	686.839 (69.1%)
	ACM-MILP	**14.484 (11.1%)**	**843.939 (0.5%)**	**350.115 (13.8%)**
$\eta = 0.1$	Random	190.813 (1071.1%)	560.655 (33.2%)	1051.451 (158.9%)
	G2MILP	11.157 (31.5%)	923.973 (10.0%)	824.986 (103.1%)
	ACM-MILP	**13.300 (18.4%)**	**842.010 (0.3%)**	**307.280 (24.3%)**
$\eta = 0.2$	Random	2651.630 (16173.7%)	393.655 (53.1%)	1515.076 (273.1%)
	G2MILP	7.159 (56.1%)	926.389 (10.4%)	976.679 (140.5%)
	ACM-MILP	**12.935 (20.6%)**	**859.528 (2.4%)**	**215.441 (46.9%)**

bipartite graphs transformed by the instances and compute the Jsnsen-Shannon (JS) divergence (Lin, 1991) to measure the distributional similarity between training and generated instances. We report the standardized similarity scores calculated by the metrics. Please refer to Appendix C.2 for the details on the mathematical formula for the similarity score.

Metric: Computational Hardness. To measure the similarity in computational hardness, we utilize the MILP solver Gurobi (Gurobi Optimization, LLC, 2023) to solve both the training and generated instances and report the average solving time and branching nodes. The computational hardness similarity is crucial in the downstream hyperparameter tuning task, which is more on our radar in this paper.

Application: Hyperparameter Tuning on Gurobi. We demonstrate the effectiveness of generated instances in reducing Gurobi's solving time via hyperparameter tuning. First, we randomly generate 50 distinct hyperparameter configurations for Gurobi and solve the training and generated instances by Gurobi with the 50 configurations. In this way, we can evaluate the solving time consistency of the training and generated instances with different solver configurations. Then, we utilize a Bayesian optimization package SMAC3 (Lindauer et al., 2022) to tune the Gurobi hyperparameters on the training and generated instances and report the solving time on the test set. Note that the generated instances with $\eta = 0.1$ are used to tune the hyperparameters.

Datasets. We conduct experiments on three synthetic datasets, embracing different MILP problem types. Specifically, the datasets include Maximum Independent Set (MIS) (Bergman et al., 2016), Set Covering (SC) (Balas & Ho, 1980) and Combinatorial Auction (CA) (Leyton-Brown et al., 2000). The datasets are generated in line with previous work (Gasse et al., 2019). In each dataset, 1,000 instances are for training, and 1,000 are for testing. Please refer to Appendix C.1 for more details on datasets.

Baselines. We compare our ACM-MILP with three baselines. The first baseline is the heuristic MILP generator Bowly (Bowly, 2019), which can generate MILP instances from scratch and control specific statistical distributions. We adjust the controllable parameters to align with the training set statistics, enabling Bowly to mimic the training set. The second is G2MILP (Geng et al., 2023), the first deep-learning model for MILP generation. The third is Random, which adopts the architecture of G2MILP with a random output. The replacement ratio η is set to 0.05, 0.1, and 0.2 to demonstrate the effect of different levels of modification.

Table 4. Solving time (second) on the test set by Gurobi with different hyperparameters. 'default' refers to the default hyperparameter without tuning. We run 5 restarts for each trial and report mean/std.

Model	MIS	SC	CA
default	**0.458±0.007**	4.222±0.044	0.926±0.004
training set	0.496±0.003	3.852±0.031	1.336±0.008
G2MILP	0.481±0.004	3.899±0.030	1.018±0.008
ACM-MILP	0.460±0.008	**3.804±0.019**	**0.871±0.004**

4.2. Results and Discussion

Instance Similarity. We generate 1,000 instances with each η on each dataset and evaluate the similarity between training and generated instances. We show the statistical distributional similarity in Table 1. Overall, ACM-MILP shows a comparative similarity score with G2MILP (higher on SC, lower on MIS, slightly lower on CA), higher than Bowly and Random. However, we can analyze the results in conjunction with the computational hardness similarity results shown in Table 2 and Table 3 to derive more information. On all three datasets, ACM-MILP shows a significantly lower error in preserving computational hardness, even on MIS and CA (with lower statistical distributional similarity). It demonstrates that ACM-MILP has more effectively learned the features of instances crucial for solving, indicating a tremendous potential to generate a richer diversity of instances while maintaining computational hardness.

Hyperparameter Tuning on Gurobi. For the first experiment, we randomly generate 50 different hyperparameter configurations for Gurobi. We solve the training and generated instances with these 50 different hyperparameter configurations. We set the time limit to 200 seconds for MIS and CA and 400 seconds for SC. In Fig. 4, we demonstrate the solving time compared with the CA training set. The closer to a straight line the points in each figure are, the stronger the correlation is. Fig. 4(a) shows that different datasets are not strongly correlated on the same configuration. Moreover, from Fig. 4(b,c,d), we can see that our ACM-MILP exhibits a stronger correlation with the original samples on hyperparameter configurations, which implies that instances generated via ACM-MILP can be more effectively utilized for hyperparameter tuning. Please refer to Appendix C.3 for more details and results.

For the second experiment, we apply the Bayesian optimization package SMAC3 to tune the hyperparameters of Gurobi. We set the number of Bayesian optimization trials to 200, i.e., sampling 200 hyperparameter configurations during tuning. To simulate the scenario with limited data availability, we randomly choose 20 original MILP instances as the training set. We tune the hyperparameters on the training set and 1,000 generated instances. After tuning, we solve the 1,000 instances in the test set with the tuned hyperparameters. The result is shown in Table 4. On SC and CA, the hyperparameter configuration tuned by our ACM-MILP-generated instances achieves the best solving time, demonstrating the effectiveness of ACM-MILP on the hyperparameter tuning task. On MIS, the default configuration yields the best result. However, our ACM-MILP still outperforms the other two models due to the volume of data over the training set and the quality of generated data over G2MILP.

Table 5. Solving time (second) by Gurobi and the relative errors w.r.t. the training sets as the reference. ($\eta = 0.1$)

Model	MIS	SC	CA
Reference	0.315	1.804	0.561
G2MILP	0.231 (26.7%)	1.912 (6.0%)	1.090 (94.3%)
ACM-MILP w/o G	0.239 (24.1%)	1.890 (4.8%)	0.693 (23.5%)
ACM-MILP w/o S	0.251 (20.3%)	1.713 (5.0%)	0.446 (20.5%)
ACM-MILP	**0.261 (17.1%)**	**1.797 (0.4%)**	**0.498 (11.2%)**

Table 6. Number of branching nodes of instances solved by Gurobi and relative errors w.r.t. the training sets as reference. ($\eta = 0.1$)

Model	MIS	SC	CA
Reference	16.294	839.354	406.107
G2MILP	11.517 (31.5%)	923.973 (10.0%)	824.986 (103.1%)
ACM-MILP w/o G	11.349 (30.3%)	897.706 (7.0%)	517.175 (27.3%)
ACM-MILP w/o S	12.250 (24.8%)	745.720 (11.2%)	211.240 (48.0%)
ACM-MILP	**13.300 (18.4%)**	**842.010 (0.3%)**	**307.280 (24.3%)**

4.3. Ablation Study

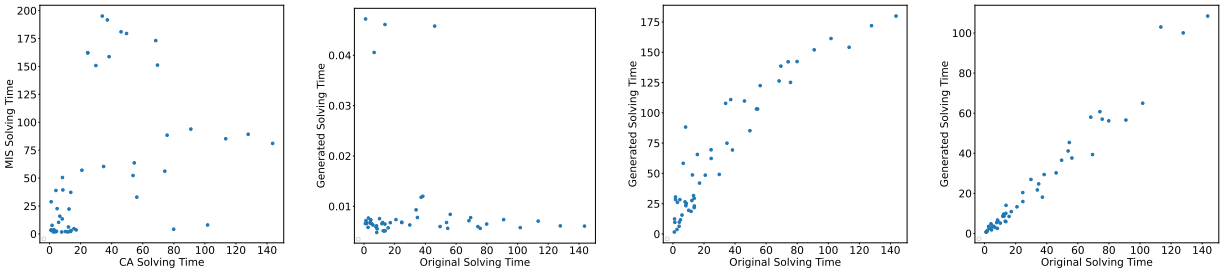
We conduct two ablation studies on the effect of constraint grouping and constraint selection, respectively. The settings of these two scenarios are as follows:

- We deactivate the constraint grouping approach by dividing each constraint node into a unique community. Thus, the model can only modify one constraint at a time. We name this approach **ACM-MILP w/o G**, which is used to evaluate the effect of constraint grouping.
- We deactivate the constraint selection approach by randomly selecting constraint communities during instance generation. We name this approach **ACM-MILP w/o S**, which is used to evaluate the effect of constraint selection.

We set the replacement ratio $\eta = 0.1$, which is a moderate value, and generate 100 instances on each dataset. The results are shown in Table 5 and Table 6, indicating that both constraint grouping and constraint selection contribute to generating hardness-preserving instances.

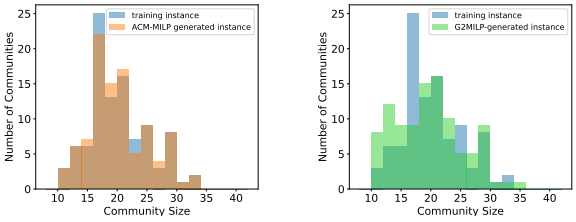
4.4. Analysis on Constraint Grouping

We further analyze the effect of our constraint grouping approach. We compare the distribution of constraint community sizes between a training and a generated instance. We show the results in Fig. 5. Our ACM-MILP includes



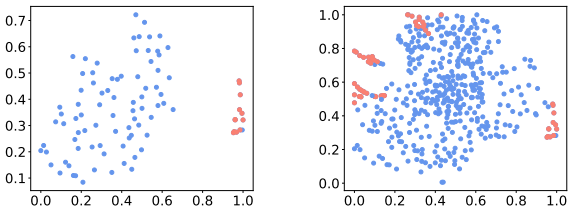
(a) MIS training set (b) Bowly (c) G2MILP, $\eta = 0.2$ (d) ACM-MILP, $\eta = 0.2$

Figure 4. Solving time (second) compared with the CA training set on 50 different hyperparameter configurations.



(a) ACM-MILP generation (b) G2MILP generation

Figure 5. Comparison of the community size distribution between the training and generated instance. Darker colors denote overlapping, and larger overlapping areas indicate closer distributions.



(a) single instance (b) instance batch

Figure 6. Visualization in the latent space: community centers in one single instance (a) and in a batch of instances (b). Red points are the communities chosen for modification.

grouping strongly related constraints, while G2MILP does not consider the interrelations among constraints. We can see that ACM-MILP-generated instance has a close distribution with the original training instance, indicating that the constraint grouping approach can preserve the interrelations among constraints. G2MILP significantly varies the community size distribution, meaning that the reconstructed constraint fails to retain the relation with other constraints.

4.5. Visualization

To better understand the effect of our constraint selection strategy, we visualize the centers of the constraint communities in the latent space in Fig. 6. It depicts that in a single instance, the communities are divided into two clusters by their sampling probabilities. The communities in the smaller cluster with lower sampling probabilities are chosen for modification. In a batch of instances, the larger clusters are close to each other and can be clustered, while the smaller clusters differ significantly. It indicates that the larger clusters mainly include generic features, and the smaller clusters contain unique features to the specific instances.

5. Conclusion

In this paper, we propose a generative framework ACM-MILP for MILP instance generation with adaptive and correlation-aware constraint reconstruction. It adopts an adaptive constraint selection mechanism based on the principle of VAE, selecting instance-unique constraints for modification and retaining generic constraints to preserve characteristics. It also detects and groups constraints with strong interrelations via community detection, allowing for collective modification. Experimental results show that ACM-MILP significantly improves the problem-solving hardness similarity over the baselines, even with more diversity. In the downstream task, we demonstrate the effectiveness of ACM-MILP-generated instances on hyperparameter tuning.

Acknowledgements

The work was partly supported by NSFC (92370201, 62222607, 72342023).

Impact Statement

This paper makes a significant contribution to the field of Mixed-integer linear programming (MILP) by addressing the critical issue of data scarcity and the need for high-quality instance generation. By introducing the ACM-MILP framework, this paper proposes a novel approach that enhances the quality and solvability of MILP instances, which is essential in real-world scenarios. This advancement is crucial for the MILP community, as it offers a method to generate instances that are more representative of actual challenges encountered in practice. The techniques to adaptively modify constraints of the framework are particularly promising, as they can be integrated with other MILP gener-

ation methods and ensure the preservation of the inherent complexity and dependencies in MILP problems. Furthermore, the demonstrated effectiveness of these generated instances in hyperparameter tuning highlights the framework's practical utility, potentially leading to more efficient and accurate MILP solvers. Overall, this paper contributes significantly to the broader MILP research community by providing a sophisticated framework for instance generation, thereby enhancing the research and application of MILP in various real-world scenarios.

References

Atamtürk, A. On the facets of the mixed–integer knapsack polyhedron. *Mathematical Programming*, 98(1-3):145–175, 2003.

Aynaud, T. python-louvain x.y: Louvain algorithm for community detection. https://github.com/taynaud/python-louvain, 2020.

Balas, E. and Ho, A. *Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study*. Springer, 1980.

Balyo, T., Froleyks, N., Heule, M. J., Iser, M., Järvisalo, M., and Suda, M. Proceedings of sat competition 2020: Solver and benchmark descriptions. 2020.

Barber, M. J. Modularity and community detection in bipartite networks. *Physical Review E*, 76(6):066102, 2007.

Bengio, Y., Lodi, A., and Prouvost, A. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

Bergman, D., Cire, A. A., Van Hoeve, W.-J., and Hooker, J. *Decision diagrams for optimization*, volume 1. Springer, 2016.

Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., Gottwald, L., Graczyk, C., Halbig, K., Hoen, A., Hojny, C., van der Hulst, R., Koch, T., Lübbecke, M., Maher, S. J., Matter, F., Mühmer, E., Müller, B., Pfetsch, M. E., Rehfeldt, D., Schlein, S., Schlösser, F., Serrano, F., Shinano, Y., Sofranac, B., Turner, M., Vigerske, S., Wegscheider, F., Wellner, P., Weninger, D., and Witzig, J. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021a. URL http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., Gottwald, L., Graczyk, C., Halbig, K., Hoen, A., Hojny, C., van der Hulst, R., Koch, T., Lübbecke, M., Maher, S. J., Matter, F., Mühmer, E., Müller, B., Pfetsch, M. E., Rehfeldt, D., Schlein, S., Schlösser, F., Serrano, F., Shinano, Y., Sofranac, B., Turner, M., Vigerske, S., Wegscheider, F., Wellner, P., Weninger, D., and Witzig, J. The SCIP Optimization Suite 8.0. ZIB-Report 21-41, Zuse Institute Berlin, December 2021b. URL http://nbn-resolving.de/urn:nbn:de:0297-zib-85309.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

Bonald, T., De Lara, N., Lutz, Q., and Charpentier, B. Scikit-network: Graph analysis in python. *Journal of Machine Learning Research*, 21(185):1–6, 2020.

Bowly, S., Smith-Miles, K., Baatar, D., and Mittelmann, H. Generation techniques for linear programming instances with controllable properties. *Mathematical Programming Computation*, 12(3):389–415, 2020.

Bowly, S. A. *Stress testing mixed integer programming solvers through new test instance generation methods*. PhD thesis, School of Mathematical Sciences, Monash University, 2019.

Brown, N., Fiscato, M., Segler, M. H., and Vaucher, A. C. Guacamol: benchmarking models for de novo molecular design. *Journal of chemical information and modeling*, 59(3):1096–1108, 2019.

Chen, X., Li, Y., Wang, R., and Yan, J. Mixsatgen: Learning graph mixing for sat instance generation. In *The Twelfth International Conference on Learning Representations*, 2024.

Chen, Z., Liu, J., Wang, X., and Yin, W. On representing mixed-integer linear programs by graph neural networks. In *The Eleventh International Conference on Learning Representations*, 2023.

Clauset, A., Newman, M. E., and Moore, C. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

Cornuéjols, G., Sridharan, R., and Thizy, J.-M. A comparison of heuristics and relaxations for the capacitated plant location problem. *European journal of operational research*, 50(3):280–297, 1991.

Cplex, I. I. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

da Silva, C. G., Figueira, J., Lisboa, J., and Barman, S. An interactive decision support system for an aggregate production planning model based on multiple criteria mixed integer linear programming. *Omega*, 34(2):167–177, 2006.

Dong, Y., Pinto, J. M., Sundaramoorthy, A., and Maravelias, C. T. Mip model for inventory routing in industrial gases supply chain. *Industrial & Engineering Chemistry Research*, 53(44):17214–17225, 2014.

Drexl, A., Nissen, R., Patterson, J. H., and Salewski, F. Progen/πx–an instance generator for resource-constrained project scheduling problems with partially renewable resources and further extensions. *European Journal of Operational Research*, 125(1):59–72, 2000.

Drugan, M. M. Instance generator for the quadratic assignment problem with additively decomposable cost function. In *2013 IEEE Congress on Evolutionary Computation*, pp. 2086–2093. IEEE, 2013.

Freund, A. and Naor, J. Approximating the advertisement placement problem. *Journal of Scheduling*, 7:365–374, 2004.

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., and Lodi, A. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32, 2019.

Geng, Z., Li, X., Wang, J., Li, X., Zhang, Y., and Wu, F. A deep instance generative framework for MILP solvers under limited data availability. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=AiEipk1X0c.

Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P., Jarck, K., Koch, T., Linderoth, J., et al. Miplib 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3): 443–490, 2021.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. Automated configuration of mixed integer programming solvers. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings 7*, pp. 186–202. Springer, 2010.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Kipf, T. N. and Welling, M. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

Leyton-Brown, K., Pearson, M., and Shoham, Y. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pp. 66–76, 2000.

Li, X., Zhu, F., Zhen, H.-L., Luo, W., Lu, M., Huang, Y., Fan, Z., Zhou, Z., Kuang, Y., Wang, Z., Geng, Z., Li, Y., Liu, H., An, Z., Yang, M., Li, J., Wang, J., Yan, J., Sun, D., Zhong, T., Zhang, Y., Zeng, J., Yuan, M., Hao, J., Yao, J., and Mao, K. Machine learning insides optverse ai solver: Design principles and applications. *arXiv preprint*, 2024.

Li, Y., Mo, Y., Shi, L., and Yan, J. Improving generative adversarial networks via adversarial learning in latent space. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 8868–8881. Curran Associates, Inc., 2022.

Li, Y., Chen, X., Guo, W., Li, X., Luo, W., Huang, J., Zhen, H.-L., Yuan, M., and Yan, J. Hardsatgen: Understanding the difficulty of hard sat formula generation and a strong structure-hardness-aware baseline. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pp. 4414–4425, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400701030. doi: 10.1145/3580305.3599837. URL https://doi.org/10.1145/3580305.3599837.

Li, Y., Guo, J., Wang, R., and Yan, J. T2t: From distribution learning in training to gradient search in testing for combinatorial optimization. In *Advances in Neural Information Processing Systems*, 2023b.

Li, Y., Shi, L., and Yan, J. Iid-gan: an iid sampling perspective for regularizing mode collapse. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence*, 2023c.

Lin, J. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., and Hutter, F. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022. URL http://jmlr.org/papers/v23/21-0888.html.

Muckstadt, J. A. and Wilson, R. C. An application of mixed-integer programming duality to scheduling thermal generating systems. *IEEE Transactions on Power Apparatus and Systems*, (12), 1968.

Nair, V., Bartunov, S., Gimeno, F., von Glehn, I., Lichocki, P., Lobov, I., O'Donoghue, B., Sonnerat, N., Tjandraatmadja, C., Wang, P., Addanki, R., Hapuarachchi, T., Keck, T., Keeling, J., Kohli, P., Ktena, I., Li, Y., Vinyals, O., and Zwols, Y. Solving mixed integer programs using neural networks, 2021.

Paulus, M. B., Zarpellon, G., Krause, A., Charlin, L., and Maddison, C. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In *International conference on machine learning*, pp. 17584–17600. PMLR, 2022.

Prouvost, A., Dumouchelle, J., Scavuzzo, L., Gasse, M., Chételat, D., and Lodi, A. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020. URL https://openreview.net/forum?id=IVc9hqgibyB.

Roling, P. C., Visser, H. G., et al. Optimal airport surface traffic planning using mixed-integer linear programming. *International Journal of Aerospace Engineering*, 2008, 2008.

Sakuma, J. and Kobayashi, S. A genetic algorithm for privacy preserving combinatorial optimization. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1372–1379, 2007.

Smith-Miles, K. and Bowly, S. Generating new test instances by evolving in instance space. *Computers & Operations Research*, 63:102–113, 2015.

Vander Wiel, R. J. and Sahinidis, N. V. Heuristic bounds and test problem generation for the time-dependent traveling salesman problem. *Transportation Science*, 29(2):167–183, 1995.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Xu, K., Li, C., Tian, Y., Sonobe, T., ichi Kawarabayashi, K., and Jegelka, S. Representation learning on graphs with jumping knowledge networks, 2018.

You, J., Wu, H., Barrett, C., Ramanujan, R., and Leskovec, J. G2sat: Learning to generate sat formulas. In *Advances in neural information processing systems*, volume 32, 2019.

Zhang, C., Ouyang, W., Yuan, H., Gong, L., Sun, Y., Guo, Z., Dong, Z., and Yan, J. Towards imitation learning to branch for mip: A hybrid reinforcement learning based sample augmentation approach. In *The Twelfth International Conference on Learning Representations*, 2023a.

Zhang, J., Liu, C., Li, X., Zhen, H.-L., Yuan, M., Li, Y., and Yan, J. A survey for solving mixed integer programming via machine learning. *Neurocomputing*, 519:205–217, 2023b.

A. Bipartite Graph Features

As described in the main paper, we represent a MILP instance as a bipartite graph $\mathcal{G} = (\mathcal{C}, \mathcal{V}, \mathcal{E})$, where $\mathcal{C}$ denotes the constrain vertex set, $\mathcal{V}$ denotes the variable vertex set, and $\mathcal{E}$ refers to the edge set. The features of $\mathcal{G}$ correspondingly contain three components, denoted as $(\mathbf{C}, \mathbf{V}, \mathbf{E})$. $\mathbf{C}$, $\mathbf{V}$ and $\mathbf{E}$ are the feature matrices of $\mathcal{C}$, $\mathcal{V}$ and $\mathcal{E}$, respectively. The detailed features and descriptions can be found in Table 7.

Table 7. Detailed constraint, variable, and edge features in the bipartite graph.

Feature	Name	Description
$\mathbf{C}$	bias	The bias value of the constraint.
	positional_encoding	The positional encoding of the constraint.
$\mathbf{V}$	objective	The coefficient of the variable in the objective function.
	is_type_binary	Whether the variable is binary.
	is_type_integer	Whether the variable is an integer.
	is_type_implicit_integer	Whether the variable is an implicit integer.
	is_type_continuous	Whether the variable is continuous.
	has_lower_bound	Whether the variable has a lower bound
	has_upper_bound	Whether the variable has an upper bound
	lower_bound	The lower bound of the variable.
	upper_bound	The upper bound of the variable.
$\mathbf{E}$	coefficient	The coefficient of the edge between the constraint and variable.

The positional encoding for constraint c_i is defined as $\frac{i}{|\mathcal{C}|}$. The goal of applying the positional encoding is to improve the discrimination of different constraints and enhance representation by GNN (Chen et al., 2023), which takes a significant role in selecting edges for the constraints.

The original features of the bipartite graph are automatically extracted via the combinatorial optimization environment Ecole (Prouvost et al., 2020). Then, we manually add positional encoding as a new feature in the constraint features.

B. Implementation Details of ACM-MILP

B.1. Community Detection

Table 8. Average community number and community size on MIS, SC and CA.

dataset	avg community num	avg community size
MIS	97.25	20.09
SC	63.57	7.87
CA	13.86	13.87

As described in Sec. 3.2.1 in the main paper, we adopt the Louvain algorithm (Blondel et al., 2008) to detect the communities in the bipartite graph. The Louvain algorithm is based on the optimization of graph modularity, which is defined in Eq. 2 in the main paper. The graph modularity is well suited for describing the interrelations among constraints since larger modularity indicates more common variables are connected.

The Louvain algorithm includes moving a vertex u_i into its neighbor's community and computing the modularity gain by:

$$\Delta Q = \frac{k_{i,in}}{2\tau} - \gamma \frac{\Sigma_{tot} \cdot k_i}{2\tau^2}, \tag{15}$$

where τ is the number of edges, $k_{i,in}$ is the number of edges connecting u_i and vertices in the community, Σ_{tot} denotes the total degree of the vertices in the community, k_i refers to the degree of u_i, and γ is the resolution parameter. Note that if γ is less than 1, the algorithm favors larger communities, and greater than 1 favors smaller communities. In our implementation, we set γ to 4 on the MIS dataset, 2 on the SC dataset, and 1 on the CA dataset. With these settings, the MILP instances in

Algorithm 1 Louvain Algorithm for Community Detection

Input: Graph $\mathcal{G}(\mathcal{C} \cup \mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{U} = \mathcal{C} \cup \mathcal{V}$ and edge set $\mathcal{E}$, resolution parameter γ.
Output: Community structure of graph $\mathcal{G}$.
1: Initialize each vertex $u \in \mathcal{U}$ as a separate community;
2: Calculate the modularity gain for moving each vertex to the community of its neighbors by Eq. 15;
3: **repeat**
4: **for** each vertex $u \in \mathcal{U}$ **do**
5: Move u to the community that results in the highest modularity gain;
6: **end for**
7: Update the community structure and recalculate the modularity gain;
8: **until** no further improvement in modularity can be achieved;
9: Aggregate the graph based on the detected communities;
10: **repeat**
11: Line 2 - 9;
12: **until** no further improvement in modularity can be achieved.

these datasets can roughly be divided into communities with suitable sizes for modification. We show the average number and size of communities on these three datasets in Table 8. We also show the procedure of the Louvain algorithm in Algo. 1. We implement the Louvain algorithm by the Python package python-louvain (Aynaud, 2020).

B.2. Derivation of Loss Function

Here we show how to derive the loss function in Eq. 7 from the training objective in Eq. 6. We first derive:

$$\begin{aligned}
\log p_{\boldsymbol{\theta}}(\mathcal{G}|\tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}}) &= \mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[\log p_{\boldsymbol{\theta}}(\mathcal{G} \mid \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})\right] \\
&= \mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[\log \frac{p_{\boldsymbol{\theta}}(\mathcal{G} \mid \mathbf{Z_C}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}}) p_{\boldsymbol{\theta}}(\mathbf{Z_C})}{q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G})} \frac{q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G})}{p_{\boldsymbol{\theta}}(\mathbf{Z_C} \mid \mathcal{G}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})}\right] \\
&= \mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[\log \frac{p_{\boldsymbol{\theta}}(\mathcal{G} \mid \mathbf{Z_C}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}}) p_{\boldsymbol{\theta}}(\mathbf{Z_C})}{q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G})}\right] + \mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[\log \frac{q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G})}{p_{\boldsymbol{\theta}}(\mathbf{Z_C} \mid \mathcal{G}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})}\right] \\
&= -\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} \mid \mathcal{G}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})) + D_{\mathrm{KL}}\left[q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G}) \| p_{\boldsymbol{\theta}}(\mathbf{Z_C} \mid \mathcal{G}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})\right] \\
&\geq -\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} \mid \mathcal{G}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}}).
\end{aligned} \tag{16}$$

In the above formula, $-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} \mid \mathcal{G}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})$ is called the (variational) lower bound, and can be written as:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi} \mid \mathcal{G}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}}) &= \mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[\log \frac{q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G})}{p_{\boldsymbol{\theta}}(\mathcal{G} \mid \mathbf{Z_C}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}}) p_{\boldsymbol{\theta}}(\mathbf{Z_C})}\right] \\
&= \mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[-\log p_{\boldsymbol{\theta}}(\mathcal{G} \mid \mathbf{Z_C}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})\right] + \mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[\log \frac{q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G})}{p_{\boldsymbol{\theta}}(\mathbf{Z_C})}\right] \\
&= \mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[-\log p_{\boldsymbol{\theta}}(\mathcal{G} \mid \mathbf{Z_C}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})\right] + D_{\mathrm{KL}}\left[q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G}) \| p_{\boldsymbol{\theta}}(\mathbf{Z_C})\right].
\end{aligned} \tag{17}$$

Note that $\tilde{\mathcal{G}}$ and $\mathbf{H}_{\setminus \mathbf{C}}$ are both decided by the modified community $\mathbf{C}$. Therefore, we take the expectation of graph $\mathcal{G}$ and community $\mathbf{C}$, and add a hyperparameter to control the weight of the KL divergence loss, thereby deriving the loss function:

$$\mathcal{L} = \mathbb{E}_{\mathcal{G} \sim \mathcal{D}} \mathbb{E}_{\mathbf{C} \sim \tilde{p}_{\boldsymbol{\phi}(\mathcal{G})}} [\underbrace{\mathbb{E}_{\mathbf{Z_C} \sim q_{\boldsymbol{\phi}}(\mathbf{Z_C}|\mathcal{G})} \left[-\log p_{\boldsymbol{\theta}}(\mathcal{G} \mid \mathbf{Z_C}, \tilde{\mathcal{G}}, \mathbf{H}_{\setminus \mathbf{C}})\right]}_{\mathcal{L}_{\mathrm{r}}} + \beta \cdot \underbrace{D_{\mathrm{KL}}\left[q_{\boldsymbol{\phi}}(\mathbf{Z_C} \mid \mathcal{G}) \| p_{\boldsymbol{\theta}}(\mathbf{Z_C})\right]}_{\mathcal{L}_{\mathrm{d}}}]. \tag{18}$$

B.3. Network Architecture

B.3.1. GNN ARCHITECTURE

As described in the main paper, we adopt a GNN to extract the features of the bipartite graph in the encoder, decoder, and graph embedding model. Here, we show the detailed architecture and procedure of the GNN.

We first yield the initial embedding of constraints $\mathbf{z}_{c_i}^{(0)}$, variables $\mathbf{z}_{v_j}^{(0)}$ and edges $\mathbf{z}_{e_{ij}}^{(0)}$ from the original features by Eq. 8. Then, we adopt K bipartite graph convolution layers. In each convolution layer, we perform the following convolutions:

$$\mathbf{z}_{c_i}^{(k+1)} = \text{MLP}\left(\mathbf{z}_{c_i}^{(k)}, \sum_{j:e_{ij}\in\mathcal{E}} \text{MLP}\left(\mathbf{z}_{c_i}^{(k)}, \mathbf{z}_{e_{ij}}^{(0)}, \mathbf{z}_{v_j}^{(k)}\right)\right) \tag{19}$$

$$\mathbf{z}_{v_j}^{(k+1)} = \text{MLP}\left(\mathbf{z}_{v_j}^{(k)}, \sum_{i:e_{ij}\in\mathcal{E}} \text{MLP}\left(\mathbf{z}_{c_i}^{(k+1)}, \mathbf{z}_{e_{ij}}^{(0)}, \mathbf{z}_{v_j}^{(k)}\right)\right). \tag{20}$$

A Graphnorm layer is followed after each convolution. After performing K bipartite graph convolution layers, we apply a Jumping Knowledge layer (Xu et al., 2018) with concatenation to derive the final vertex representations:

$$\mathbf{z}_u^{(\text{final})} = \text{MLP}\left(\left[\mathbf{z}_u^{(0)}, \mathbf{z}_u^{(1)}, \cdots, \mathbf{z}_u^{(K)}\right]\right), \quad u \in \mathcal{C}\cup\mathcal{V}. \tag{21}$$

B.4. Decoder Modules

In the main paper, we elaborate on the procedure of the decoding phase. A GNN is first employed to extract the refined vertex features $\mathbf{p}_u$ after merging by Eq. 14. Then, we design five modules to accomplish the reconstruction of a constraint community. Here, we present the detailed architecture of the five decoder modules.

Bias Predictor. We apply an MLP to predict the normalized bias of each constraint in $\mathbf{C}$ by:

$$\hat{b}_{c_i}^{\text{pred}} = \text{Sigmoid}\left(\text{MLP}_{\boldsymbol{\theta}}^{b}(\mathbf{p}_{c_i})\right), \quad c_i \in \mathbf{C}, \tag{22}$$

where the Sigmoid function is adopted to restrict the output range to $[0, 1]$. The final predicted bias $b_{c_i}^{\text{pred}}$ is computed via $b_{c_i}^{\text{pred}} = (b_{\max} - b_{\min})\hat{b}_{c_i}^{\text{pred}} + b_{\min}$, where $b_{\max}$ and $b_{\min}$ are respectively the maximum and minimum bias that occur in the dataset. The bias predictor is trained by MSE loss.

Degree Predictor. Similar to the bias predictor, we apply an MLP to predict the normalized degree via:

$$\hat{d}_{c_i}^{\text{pred}} = \text{Sigmoid}\left(\text{MLP}_{\boldsymbol{\theta}}^{d}(\mathbf{p}_{c_i})\right), \quad c_i \in \mathbf{C}. \tag{23}$$

The final predicted degree $d_{c_i}^{\text{pred}}$ is calculated through $d_{c_i}^{\text{pred}} = (d_{\max} - d_{\min})\hat{d}_{c_i}^{\text{pred}} + d_{\min}$, where $d_{\max}$ and $d_{\min}$ are respectively the maximum and minimum degree that occur in the dataset. The degree predictor is also trained by MSE loss.

Variable Pool Predictor. Denote the pool containing the connected variables with the community $\mathbf{C}$ as $\mathcal{P}_{\mathbf{C}}$. We consider predicting whether a variable is in $\mathcal{P}_{\mathbf{C}}$ as a binary classification task and adopt an MLP with the input of variable vertex feature $\mathbf{p}_{v_i}$:

$$\xi_{v_i} = \text{Sigmoid}\left(\text{MLP}_{\boldsymbol{\theta}}^{v}(\mathbf{p}_{v_i})\right), \quad v_i \in \mathcal{V}, \tag{24}$$

where ξ_{v_i} denotes the probability that v_i is in $\mathcal{P}_{\mathbf{C}}$. In training, ξ_{v_i} is used to compute the BCE loss to train the predictor. For inference, the variables with top-k probabilities are preserved in $\mathcal{P}_{\mathbf{C}}$, where k is the sum of $d_{c_i}^{\text{pred}}$ for $c_i \in \mathbf{C}$.

Edge Selector. It selects the edges between a constraint in community $\mathbf{C}$ and variables in $\mathcal{P}_{\mathbf{C}}$ through an MLP:

$$\xi_{c_i,v_j} = \text{Sigmoid}\left(\text{MLP}_{\boldsymbol{\theta}}^{e}([\mathbf{p}_{c_i}, \mathbf{p}_{v_j}])\right), \tag{25}$$

where $c_i \in \mathbf{C}$ and $v_j \in \mathcal{P}_{\mathbf{C}}$. $[\cdot]$ denotes the concatenation operation. ξ_{c_i,v_j} refers to the probability of an existing edge between c_i and v_j. This module is trained by BCE loss. During inference, we select the edges for a constraint by sampling with the probability $\xi_{c_i,\sim}$, rather than directly choosing the variables with top-k probabilities. This approach helps avoid generating duplicated constraints.

Edge Weight Predictor. Finally, an MLP is applied to predict the normalized edge weight by:

$$\hat{w}_{c_i,v_j} = \text{Sigmoid}\left(\text{MLP}^{w}_{\boldsymbol{\theta}}([\mathbf{p}_{c_i}, \mathbf{p}_{v_j}])\right), \tag{26}$$

ensuring that an edge exists between c_i and v_j. The final predicted weight is computed in a similar way to the bias through $w^{\text{pred}}_{c_i,v_j} = (w_{\max} - w_{\min})\hat{w}_{c_i,v_j} + w_{\min}$, where $w_{\max}$ and $w_{\min}$ are respectively the maximum and minimum constraint coefficient that occur in the dataset. This module is trained by MSE loss.

The losses from these five modules collectively constitute the reconstruction loss $\mathcal{L}_{\text{r}}$, which is jointly optimized with the divergence loss $\mathcal{L}_{\text{d}}$ during training.

More details on the choices of model architecture hyperparameters can be found in our source code.

B.5. Training and Inference Procedure

Here, we demonstrate the training and inference procedures of ACM-MILP in Algo. 2 and Algo. 3, respectively.

Algorithm 2 Training procedure of ACM-MILP

Input: Dataset $\mathcal{D}$, batch size B, number of training steps N_1 for stage 1, number of training steps N_2 for stage 2, size of community pool s for fine-tuning in stage 2.
Output: Trained ACM-MILP.

Detect communities for each instance in $\mathcal{D}$ by Algo. 1; ▷ coupling constraints
for $i = 1, \cdots, N_1$ **do**
  $\mathcal{B} \leftarrow \emptyset$;
  for $b = 1, \cdots, B$ **do**
    $\mathcal{G} \sim \mathcal{D}$, $\mathbf{C} \sim \mathbf{C}_{\mathcal{G}}$; ▷ randomly select graph and community
    $\mathcal{B} \leftarrow \{(\mathcal{G}, \mathbf{C})\}$;
    for $c_i \in \mathbf{C}$ **do**
      Compute $b^{\text{pred}}_{c_i}$, $d^{\text{pred}}_{c_i}$, ξ_v, $\xi_{c_i,v}$, $w^{\text{pred}}_{c_i,v}$ with parameters $\boldsymbol{\phi}$, $\boldsymbol{\theta}$, $\boldsymbol{\mu}$;
    end for
    Compute $\mathcal{L}_{\mathcal{G},\mathbf{C}}$ by Eq. 17;
  end for
  $\mathcal{L} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{(\mathcal{G},\mathbf{C})\in\mathcal{B}} \mathcal{L}_{\mathcal{G},\mathbf{C}}$;
  Update $\boldsymbol{\phi}$, $\boldsymbol{\theta}$, $\boldsymbol{\mu}$ to minimize $\mathcal{L}$; ▷ training in stage 1
end for
for $i = 1, \cdots, N_2$ **do**
  $\mathcal{B} \leftarrow \emptyset$;
  for $b = 1, \cdots, B$ **do**
    $\mathcal{G} \sim \mathcal{D}$;
    Select s communities $\mathbf{C}_{\mathcal{G},s}$ with top-s lowest Gaussian sampling probabilities from $\mathbf{C}_{\mathcal{G}}$;
    for $\mathbf{C} \in \mathbf{C}_{\mathcal{G},s}$ **do**
      $\mathcal{B} \leftarrow \{(\mathcal{G}, \mathbf{C})\}$;
      for $c_i \in \mathbf{C}$ **do**
        Compute $b^{\text{pred}}_{c_i}$, $d^{\text{pred}}_{c_i}$, ξ_v, $\xi_{c_i,v}$, $w^{\text{pred}}_{c_i,v}$ with parameters $\boldsymbol{\phi}$, $\boldsymbol{\theta}$, $\boldsymbol{\mu}$;
      end for
      Compute $\mathcal{L}_{\mathcal{G},\mathbf{C}}$ by Eq. 17;
    end for
  end for
  $\mathcal{L} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{(\mathcal{G},\mathbf{C})\in\mathcal{B}} \mathcal{L}_{\mathcal{G},\mathbf{C}}$;
  Update $\boldsymbol{\phi}$, $\boldsymbol{\theta}$, $\boldsymbol{\mu}$ to minimize $\mathcal{L}$; ▷ fine-tuning in stage 2
end for
return the detected community structure

B.6. Testbed

All experiments are done on our workstation with AMD 3970X, RTX 3090, and 128GB memory.

Algorithm 3 Generate a new MILP instance with ACM-MILP

Input: Dataset $\mathcal{D}$, trained ACM-MILP, replacement ratio η.
Output: A new MILP instance $\hat{\mathcal{G}}$.

$\mathcal{G} \sim \mathcal{D}$, $s \leftarrow \lceil \eta \cdot |\mathcal{C}_{\mathcal{G}}| \rceil$; ▷ sample an instance, and compute the number of communities to be modified
Select s communities $\mathbf{C}_{\mathcal{G},s}$ with top-s lowest Gaussian sampling probabilities from $\mathbf{C}_{\mathcal{G}}$;
for $\mathbf{C} \in \mathbf{C}_{\mathcal{G},s}$ **do**
  Compute $b^{\text{pred}}_{c_i}$, $d^{\text{pred}}_{c_i}$ for all $c_i \in \mathbf{C}$ with ACM-MILP;
  Compute ξ_v for $v \in \mathcal{V}_{\mathcal{G}}$ with ACM-MILP;
  Construct the connected variable pool $\mathcal{V}_{\text{connect}}$;
  for $c_i \in \mathbf{C}$ **do**
    Compute $\xi_{c_i,v}$ for $v \in \mathcal{V}_{\text{connect}}$ with ACM-MILP;
    Sample edges for $d^{\text{pred}}_{c_i}$ times with sampling probability $\xi_{c_i,v}$ for $v \in \mathcal{V}_{\text{connect}}$;
    Compute $w^{\text{pred}}_{c_i,v}$ for selected edges;
    Replace the original bias, edges, and edge weights of c_i with the newly generated ones.
  end for
end for

C. Details on Experiments

C.1. Dataset Statistics

As described in the main paper, we conduct experiments on three datasets, including Maximum Independent Set (MIS), Set Covering (SC), and Combinatorial Auction (CA). We demonstrate the detailed information of these datasets in Table 9.

Table 9. The statistics of datasets. # of training refers to the number of instances in the training set, and # of the test denotes the number of instances in the test set. $|\mathcal{C}|$ and $|\mathcal{V}|$ mean the average number of constraints and variables, respectively.

dataset	# of training	# of test	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{V}\rvert$
MIS	1000	1000	1954	500
SC	1000	1000	500	1000
CA	1000	1000	192	500

C.2. Calculations of Statistical Distributional Similarity Score

We use 11 graph statistics to evaluate the statistical distributional similarity. We show the used statistics in Table 10. We first calculate the statistics for each training and generated instance. Then, we estimate the distributions and the cross entropy by functions in the Python package *numpy* and *scipy*, respectively. We compute the JS divergence $D_{\text{JS},i}$ for the i-th statistic between the training and generated instances, and standardize $D_{\text{JS},i}$ by:

$$D^{\text{norm}}_{\text{JS},i} = \frac{1}{\log 2}(\log 2 - D_{\text{JS},i}). \tag{27}$$

The statistical distributional similarity score is the mean of all normed JS divergences, and is calculated by:

$$\text{score} = \frac{1}{11}\sum_{i=1}^{11} D^{\text{norm}}_{\text{JS},i}, \tag{28}$$

ranging from 0 to 1. A higher score implies higher similarity in distribution.

C.3. Downstream Tasks

Solving with Different Hyperparameters As described in the main paper, we conduct experiments on hyperparameter tuning on Gurobi. We first solve the training and generated instances with 50 distinct hyperparameter configurations and measure the consistency of the training and generated instances in solving with different solver configurations. This

Table 10. The 11 graph statistics used for measuring statistical distributional similarity.

Feature	Description
coef_dens	Proportion of non-zero elements in matrix $\boldsymbol{A}$, calculated as $\lvert\mathcal{E}\rvert/(\lvert\mathcal{V}\rvert \cdot \lvert\mathcal{W}\rvert)$.
cons_degree_mean	Average degree of constraint vertices in $\mathcal{V}$.
cons_degree_std	Standard deviation of constraint vertex degrees in $\mathcal{V}$.
var_degree_std	Standard deviation of variance vertex degrees in $\mathcal{W}$.
lhs_mean	Average value of non-zero elements in matrix $\boldsymbol{A}$.
lhs_std	Standard deviation of non-zero elements in matrix $\boldsymbol{A}$.
rhs_mean	Average value of elements in vector $\boldsymbol{b}$.
rhs_std	Standard deviation of elements in vector $\boldsymbol{b}$.
clustering_coef	Clustering coefficient of the graph.
modularity	Modularity of the graph.

Table 11. Selected hyperparameters of Gurobi. 'category' refers to the component that the hyperparameter affects in solving process.

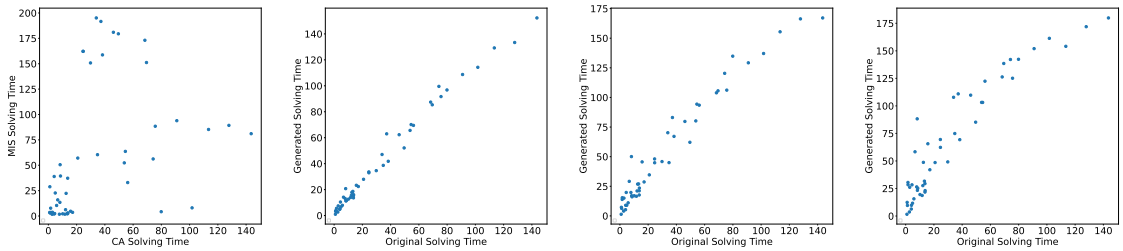
hyperparameter	category	value type	selecting range	description
Heuristics	MIP	double	[0, 1]	Turn MIP heuristics up or down.
MIPFocus	MIP	integer	{0, 1, 2, 3}	Set the focus of the MIP solver.
VarBranch	MIP	integer	{-1, 0, 1, 2, 3}	Branch variable selection strategy.
BranchDir	MIP	integer	{-1, 0 ,1}	Branch direction preference.
RINS	MIP	integer	{-1, 0, $\cdots$, 20}	RINS heuristic.
PartitionPlace	MIP	integer	{0, 1, $\cdots$, 31}	Controls when the partition heuristic runs.
NodeMethod	MIP	integer	{-1, 0, 1, 2}	Method used to solve MIP node relaxations.
LPWarmStart	Simplex	integer	{0, 1, 2}	Warm start usage in simplex.
PerturbValue	Simplex	double	[0, 0.001]	Simplex perturbation magnitude.
Presolve	Presolve	integer	{-1, 0, 1, 2}	Presolve level.
Prepasses	Presolve	integer	{-1, 0, $\cdots$, 20}	Presolve pass limit.
Cuts	MIP Cuts	integer	{-1, 0, 1, 2, 3}	Global cut generation control.
CliqueCuts	MIP Cuts	integer	{-1, 0, 1, 2}	Clique cut generation.
CoverCuts	MIP Cuts	integer	{-1, 0, 1, 2}	Cover cut generation.
Method	Other	integer	{-1, 0, 1, 2, 3, 4, 5}	Algorithm used to solve continuous models.

experiment shows the consistency of tuning hyperparameters with generated and training instances. Here, we demonstrate the hyperparameters of Gurobi related to this experiment in Table 11. In our experiment, we set the random seed to 0, $\cdots$, 50 each time, and randomly select each hyperparameter from the selecting range via the Python package 'random'. The hyperparameters fall into various categories, including cut, branch, presolve, and LP relaxation, thereby significantly affecting the solving process.

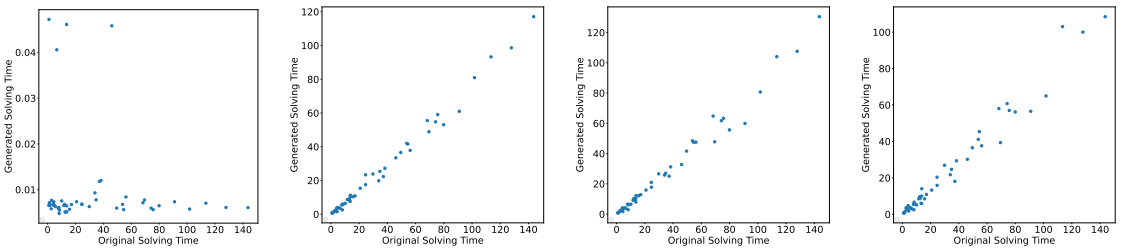
Due to space limitations, we only demonstrate the results on CA in the main paper. Here, we show full results on CA, MIS, and SC in Fig. 7, Fig. 8 and Fig. 9, respectively. We also calculate the Pearson Correlation Coefficients (PCC) and the log p-values of solving time and report them in Table 12. We can see that our method outperforms G2MILP in most cases.

Hyperparameter Tuning on Gurobi We conduct the hyperparameter tuning experiment on Gurobi. We compare the solving time of Gurobi utilizing the following configurations on 1,000 instances in the test set:

- 'default': The default configuration of hyperparameters without tuning.
- 'training set': The configuration of hyperparameters tuned on 20 instances selected in the training set. We only utilize 20 instances to simulate the situation of data scarcity.



(a) MIS training set (b) G2MILP, $\eta = 0.05$ (c) G2MILP, $\eta = 0.1$ (d) G2MILP, $\eta = 0.2$



(e) Bowly (f) ACM-MILP, $\eta = 0.05$ (g) ACM-MILP, $\eta = 0.1$ (h) ACM-MILP, $\eta = 0.2$

Figure 7. Solving time (second) compared with the CA training set on 50 different hyperparameter configurations.

Table 12. Pearson Correlation Coefficients (PCC) and the log p-values of solving time between training and generated instances.

Replacement Ratio	Model	MIS		CA		SC	
		PCC ↑	log p-value ↓	PCC ↑	log p-value ↓	PCC ↑	log p-value ↓
/	Bowly	-0.068	0.639	-0.127	0.378	0.042	0.772
$\eta = 0.05$	G2MILP	0.995	-49.332	0.991	-42.467	0.989	-40.545
	ACM-MILP	**0.997**	**-51.630**	**0.995**	**-47.511**	**0.997**	**-54.524**
$\eta = 0.1$	G2MILP	0.988	**-39.721**	0.976	-31.550	0.997	-53.443
	ACM-MILP	0.988	-38.966	**0.991**	**-41.494**	**0.998**	**-56.987**
$\eta = 0.2$	G2MILP	0.970	-29.593	0.934	-21.399	0.993	-46.017
	ACM-MILP	**0.990**	**-40.944**	**0.985**	**-36.692**	**0.998**	**-56.074**

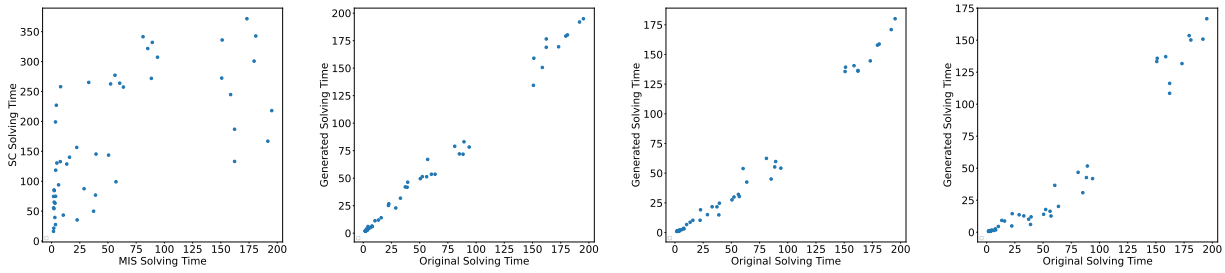
- 'G2MILP': The configuration of hyperparameters tuned on 1,000 instances generated by G2MILP. We set $\eta = 0.1$.
- 'ACM-MILP': The configuration of hyperparameters tuned on 1,000 instances generated by ACM-MILP. We set $\eta = 0.1$.

We tune the hyperparameters by Bayesian optimization. We employ the Python Bayesian optimization package SMAC3 and run 200 trials, which means sampling and evaluating 200 configurations in each tuning process. To achieve a trade-off between tuning time and tuning effectiveness, we select eight major hyperparameters from Table 11, including Heuristics, MIPFocus, VarBranch, BranchDir, Presolve, PrePasses, Cuts, and Method. The selecting ranges of hyperparameters remain the same as Table 11.

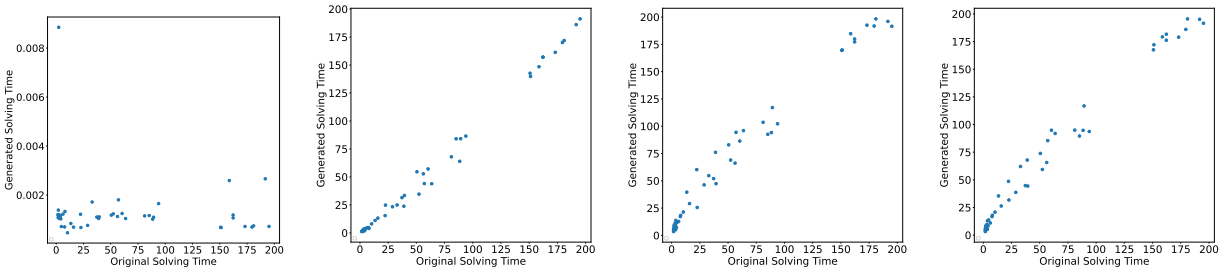
D. Further Experiments and Analysis

D.1. Comparison with Bipartite Modularity

When adopting the Louvain algorithm for community detection in the main paper, we use the standard modularity definition (Clauset et al., 2004). However, there is a modularity metric designed for bipartite graph (Barber, 2007). We implement the bipartite Louvain algorithm, i.e., the Louvain algorithm on the bipartite modularity definition, with the Python package scikit-network (Bonald et al., 2020). We detect the communities on the training set of MIS, SC, and CA via the bipartite

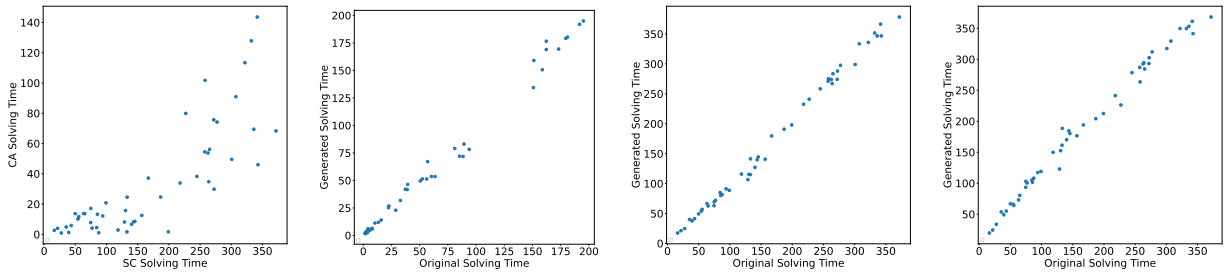


(a) SC training set (b) G2MILP, $\eta = 0.05$ (c) G2MILP, $\eta = 0.1$ (d) G2MILP, $\eta = 0.2$

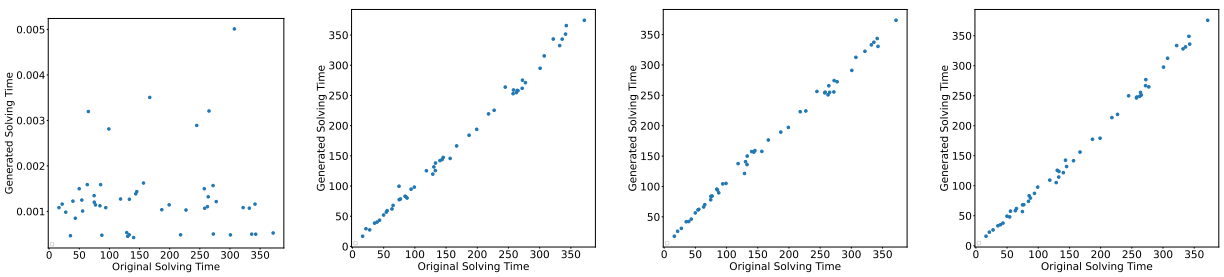


(e) Bowly (f) ACM-MILP, $\eta = 0.05$ (g) ACM-MILP, $\eta = 0.1$ (h) ACM-MILP, $\eta = 0.2$

Figure 8. Solving time (second) compared with the MIS training set on 50 different hyperparameter configurations.



(a) CA training set (b) G2MILP, $\eta = 0.05$ (c) G2MILP, $\eta = 0.1$ (d) G2MILP, $\eta = 0.2$



(e) Bowly (f) ACM-MILP, $\eta = 0.05$ (g) ACM-MILP, $\eta = 0.1$ (h) ACM-MILP, $\eta = 0.2$

Figure 9. Solving time (second) compared with the SC training set on 50 different hyperparameter configurations.

Table 13. Average modularity on the training set. 'Algorithm' refers to the algorithm adopted for community detection.

Algorithm	MIS	SC	CA
Ordinary Louvain	0.629	0.148	0.344
Bipartite Louvain	0.608	0.114	0.342

Table 14. Solving time (second) by Gurobi and the relative errors w.r.t. the training sets. We run 5 restarts for each trial and report mean/std. ($\eta = 0.1$)

Algorithm	Model	MIS	SC	CA
/	/ (training set)	0.315±0.002	1.804±0.013	0.561±0.002
Ordinary Louvain	ACM-MILP	0.261±0.005 (17.1%)	**1.797±0.004 (0.4%)**	**0.498±0.001 (11.2%)**
Bipartite Louvain	ACM-MILP	**0.271±0.006 (14.0%)**	1.875±0.009 (3.9%)	0.491±0.002 (12.5%)

Table 15. Number of branching nodes of instances solved by Gurobi and the relative errors w.r.t. the training sets. ($\eta = 0.1$)

Algorithm	Model	MIS	SC	CA
/	/ (training set)	16.294	839.354	406.107
Ordinary Louvain	ACM-MILP	13.300 (18.4%)	**842.010 (0.3%)**	**307.280 (24.3%)**
Bipartite Louvain	ACM-MILP	**13.980 (14.2%)**	907.780 (8.2%)	293.740 (27.7%)

Table 16. Solving time (second) by Gurobi and the relative errors w.r.t. the training sets. We run 5 restarts for each trial and report mean/std.

Replacement Ratio	Model	MIS	SC	CA
0 (Training set)	/	0.315±0.002	1.804±0.013	0.561±0.002
$\eta = 0.3$	G2MILP	0.114±0.001 (63.8%)	1.426±0.021 (21.0%)	1.908±0.010 (240.1%)
	ACM-MILP	**0.214±0.004 (32.1%)**	**1.774±0.017 (1.7%)**	**0.436±0.004 (22.3%)**
$\eta = 0.4$	G2MILP	0.092±0.002 (70.8%)	1.259±0.022 (30.2%)	1.856±0.014 (230.8%)
	ACM-MILP	**0.168±0.003 (46.7%)**	**2.086±0.026 (15.6%)**	**0.387±0.003 (31.0%)**
$\eta = 0.5$	G2MILP	0.070±0.001 (77.8%)	2.446±0.028 (35.6%)	1.751±0.008 (212.1%)
	ACM-MILP	**0.103±0.002 (67.3%)**	**1.645±0.020 (8.8%)**	**0.311±0.004 (44.6%)**

Table 17. Number of branching nodes of instances solved by Gurobi and the relative errors w.r.t. the training sets.

Replacement Ratio	Model	MIS	SC	CA
0 (Training set)	/	16.294	839.354	406.107
$\eta = 0.3$	G2MILP	5.312 (67.4%)	391.921 (53.3%)	1130.160 (178.3%)
	ACM-MILP	**8.560 (47.5%)**	**625.343 (25.5%)**	**217.000 (46.6%)**
$\eta = 0.4$	G2MILP	5.660 (65.3%)	323.282 (61.5%)	1149.630 (183.1%)
	ACM-MILP	**7.483 (54.1%)**	**892.280 (6.3%)**	**125.240 (69.2%)**
$\eta = 0.5$	G2MILP	6.281 (61.5%)	1357.380 (61.7%)	1112.340 (173.9%)
	ACM-MILP	**6.866 (57.9%)**	**624.610 (25.6%)**	**73.760 (81.8%)**

Louvain algorithm and compare the difference between the modularities computed by ordinary Louvain and bipartite Louvain. The results are demonstrated in Table 13. The modularities derived by the two algorithms are slightly different. Furthermore, we conduct MILP instance generation experiments to evaluate the effect of bipartite Louvain. We set the replacement ratio $\eta = 0.1$ and generate 1,000 instances on each dataset. The results are demonstrated in Table 14 and Table 15. The two approaches show similar performance.

D.2. Experiments with More Values of η

To further verify the effectiveness of our ACM-MILP, we conduct experiments with more values of η, and demonstrate the results in Table 16 and Table 17. The results show the effectiveness of our model on multiple values of η.

Table 18. Solving time (second) and number of branching nodes by Gurobi and the relative errors w.r.t. the training sets on MIS. We run 5 restarts for each trial and report mean/std. ($\eta = 0.1$)

Selection Scheme	Model	Solving Time	Branching Nodes
High Prob	ACM-MILP	0.151±0.002 (52.1%)	3.57 (78.5%)
Low Prob	ACM-MILP	**0.261±0.005 (17.1%)**	**13.30 (18.4%)**

Table 19. Number and ratio of variables taking different values in the solutions between the original and generated instances.

Replacement Ratio	Difference Number	Difference Ratio
$\eta = 0.01$	28	5.6%
$\eta = 0.05$	57	11.4%
$\eta = 0.1$	78	15.6%
$\eta = 0.2$	141	28.2%

D.3. Further Analysis on Constraint Selection

In this section, we will provide insights into the importance of selecting constraint communities with low sampling probabilities. We supplement two rationales for this design with empirical evidence presented below:

- **Communities with high sampling probabilities should be maintained to preserve the solving characteristics of MILP instances.** As claimed in HardSATGEN (Li et al., 2023a), combinatorial optimization instances' core logical structures, which may correspond to high sampling probability in this context, are challenging for predictive neural networks to learn. We supplement experiments on the MIS dataset, where we modify the communities with high sampling probabilities with the same paradigm for comparison. The results in Table 18 indicate that modifying the communities with high sampling probabilities leads to significant quality degradation of generated samples. This further suggests that the structures of these communities cannot be effectively learned by neural networks, and thus should be maintained.
- **Communities with low sampling probabilities are not noisy and uninformative constraints and maintain a clear impact on the solution.** We solve the original MIS instances and the corresponding generated MIS instances, where all variables are binary. Subsequently, we perform a statistical analysis on the number and ratio of variables taking different values in the solutions produced by Gurobi between the original and the generated instances. The results are demonstrated in Table 19. It can be seen that even if the top 1% of communities with the lowest sampling probabilities are modified, the solution can be clearly influenced. It indicates that communities with low sampling probabilities are also important substructures for learning and generating.