Mixed-Integer Linear Optimization via Learning-Based Two-Layer Large Neighborhood Search

Wenbo Liu[1], Akang Wang[2,3], Wenguo Yang[1] ⋆, and Qingjiang Shi[2,4]

[1] University of Chinese Academy of Sciences, Beijing, China
[2] Shenzhen International Center for Industrial and Applied Mathematics, Shenzhen Research Institute of Big Data, China
[3] The Chinese University of Hong Kong, Shenzhen, China
[4] Tongji University, Shanghai, China

Abstract. Mixed-integer linear programs (MILPs) are extensively used to model practical problems such as planning and scheduling. A prominent method for solving MILPs is large neighborhood search (LNS), which iteratively seeks improved solutions within specific neighborhoods. Recent advancements have integrated machine learning techniques into LNS to guide the construction of these neighborhoods effectively. However, for large-scale MILPs, the search step in LNS becomes a computational bottleneck, relying on off-the-shelf solvers to optimize auxiliary MILPs of substantial size. To address this challenge, we introduce a two-layer LNS (TLNS) approach that employs LNS to solve both the original MILP and its auxiliary MILPs, necessitating the optimization of only small-sized MILPs using off-the-shelf solvers. Additionally, we incorporate a lightweight graph transformer model to inform neighborhood design. We conduct extensive computational experiments using public benchmarks. The results indicate that our learning-based TLNS approach achieves remarkable performance gains–up to 66% and 96% over LNS and state-of-the-art MILP solvers, respectively.

Keywords: Large neighborhood search· Mixed-integer linear programs· Graph neural networks· Learn to optimize

1 Introduction

Mixed-integer linear programs (MILPs) have become a cornerstone in various industrial applications, including network design [17], production planning [19], and route optimization [24]. The resolution of MILPs typically poses $\mathcal{NP}$-hard challenges, with general-purpose MILP solvers resorting to the branch-and-bound method for systematic enumeration of candidate solutions. However, tackling large-scale MILPs with a branch-and-bound algorithm proves computationally demanding due to its exhaustive search nature. In practical scenarios, primal

⋆ Corresponding Author: Wenguo Yang <yangwg@ucas.ac.cn>

heuristic methods are commonly employed to efficiently identify high-quality feasible solutions. Although these heuristics do not guarantee optimality, they consistently deliver outstanding solutions for significantly larger MILPs.

One of the most prominent heuristic methods for addressing MILPs is *large neighborhood search* (LNS) [21]. LNS often refines an incumbent solution by iteratively constructing a neighborhood of interest and searching within such a region via optimization of auxiliary MILPs. Lots of efforts [2,4,5,11,20] have been devoted to building neighborhoods by use of heuristics, delegating the search step to off-the-shelf MILP solvers.

Recently, *machine learning* (ML) techniques have been extensively utilized to expedite the optimization of MILPs [6,18,30]. These endeavors have been prompted by the recognition that MILPs arising from similar applications often exhibit recurrent patterns, which can be effectively captured through ML techniques. LNS also benefits from ML techniques [12,22,23,29], where neighborhood construction is informed by utilizing *graph neural networks* (GNNs) on the graph representation of MILPs. Despite promising advancements, these learning-based LNS methods still have ample room for improvement. On one hand, while neighborhood construction shows potential for enhancement, it often overlooks improvements in the search step. On the other hand, the number of GNN layers (typically 2) is limited due to issues with over-smoothing [15], rendering these applications incapable of effective message passing between distant variable nodes.

Previous efforts, in both classic and learning-enhanced settings, focused on designing effective neighborhoods, leaving the search procedure to off-the-shelf MILP solvers. However, relying on exact solvers to optimize auxiliary MILPs during the search stage could still be computationally expensive for large-sized problems. Recognizing that searching for high-quality solutions still involves optimizing MILPs, we adopt a learning-based LNS approach once more. In our method, we propose applying learning-based LNS to tackle both the original MILP and its auxiliary MILPs, which necessitates optimizing only small-sized sub-MILPs using off-the-shelf solvers. Additionally, we employ a lightweight *graph transformer* model to expand the receptive field of GNNs. We call this method "learning-based Two-Layer LNS" (TLNS). The distinct contributions of our work can be summarized as follows.

- We introduce a novel TLNS algorithm designed to identify high-quality solutions for MILPs. Unlike traditional approaches, TLNS applies LNS to optimize not only the original MILP (outer layer) but also its auxiliary MILPs (inner layer), employing a divide-and-conquer strategy. To the best of our knowledge, this represents the first attempt to extend LNS to a multi-layer version specifically tailored for addressing MILPs.
- We employ a lightweight graph transformer model trained with *contrastive loss* to effectively guide the construction of neighborhoods in LNS, furthermore boosting the algorithmic performance.

– We conduct extensive computational experiments on public benchmarks and the results show that TLNS achieves up to 66% and 96% improvements over LNS and the state-of-the-art MILP solvers, respectively.

1.1 Related Works

Traditional LNS [20] proposed to use a mutation neighborhood by fixing a random subset of integer variables at the incumbent. [5] introduced the *local branching* (LB) method, defining a neighborhood as a Hamming ball around the incumbent solution, while [11] proposed to utilize solutions to continuous relaxations in LB for building neighborhoods. RINS [4] constructs a promising neighborhood using information contained in the continuous relaxation of an MILP as well as the incumbent, while RENS [2] relies purely on the continuous relaxation. Though many aforementioned LNS heuristics have been deployed within MILP solvers, their computational effort makes it impractical to apply all of them frequently. One exception is the work of [10] in which the author considered eight popular LNS heuristics and proposed to adaptively select one of them for execution via online learning.

Learning-based LNS The first attempt to enhance LNS with ML is [22], in which the authors proposed to imitate the best neighborhood out of a few randomly sampled ones. Building upon this, [23] improved the imitation learning approach by employing LB as an expert. Furthermore, [12] collected both positive and negative solution samples from LB and then utilized contrastive loss to learn and construct neighborhoods. Alternatively, *reinforcement learning* (RL) was applied by [29] to explore a promising policy for constructing neighborhoods while [16] focused on enhancing LB and utilized RL to inform the radius of the Hamming ball.

2 Preliminaries

Mixed-Integer Linear Programs An MILP is formulated as :

$$\min_{x \in S} \quad c^\top x \tag{1}$$

where x denotes the decision variable and $S \coloneqq \{x \in \mathbb{Z}^q \times \mathbb{R}^{n-q} : Ax \leq b, l \leq x \leq u\}$ represents the feasible region for x. $l, u, c \in \mathbb{R}^n, b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$ are given parameters. For the sake of simplicity, we assume that all variables are binary, i.e., $x_i \in \{0, 1\}\ \forall i = 1, 2, ..., n$. Let $M := (A, b, c, l, u, q)$ denote an MILP instance for convenience.

Large Neighborhood Search In this work, we focus on *fixing neighborhood LNS heuristics* [10] in which neighborhoods are defined by fixing part of decision variables.

Definition 1 (Fixing neighborhood). *Consider an MILP with n variables, let $\mathcal{F} \subseteq \{1, ..., n\}$ denote an index set and $\bar{x}$ denote a reference point. Then a fixing neighborhood of $\bar{x}$ is defined by fixing variables in $\mathcal{F}$ to their values in $\bar{x}$:* $\mathcal{B}(\mathcal{F}, \bar{x}) \coloneqq \{x \in \mathbb{R}^n : x_i = \bar{x}_i, \forall i \in \mathcal{F}\}$.

The number of unfixed variables (aka *neighborhood size*) is equal to $n - |\mathcal{F}|$. Using such a fixing neighborhood, we can define an auxiliary problem.

Definition 2 (Auxiliary problem). *Given an MILP M of form (1) and a fixing neighborhood $\mathcal{B}(\mathcal{F}, \bar{x})$, then an auxiliary problem $\mathcal{A}(M, \bar{x}, \mathcal{F})$ is defined as the following MILP:*

$$\min_{x \in S \cap \mathcal{B}(\mathcal{F}, \bar{x})} \quad c^\top x. \tag{2}$$

Problem (2) guarantees feasibility of returned solutions whenever $\bar{x} \in S$. In LNS, an incumbent $\bar{x}$ is consistently considered as the reference point and will be iteratively refined by constructing a fixing neighborhood and invoking an off-the-shelf solver to optimize the auxiliary problem (2).

Bipartite Graph Representation Given an MILP of form (1), [7] proposed a variable-constraint bipartite graph representation, as shown in 1. Specifically, let $G \coloneqq (\mathcal{V}, \mathcal{E})$ denote a bipartite graph, where $\mathcal{V} \coloneqq \{v_1, ..., v_n, v_{n+1}, ..., v_{n+m}\}$ denotes the set of n variable nodes and m constraint nodes, and $\mathcal{E}$ represents the set of edges that only connect between nodes of different types. Variable node v_i and constraint node v_{n+j} are connected if A_{ji} is non-zero. The information of an MILP including c, l, u, b and A will be properly incorporated into G as graph attributes.

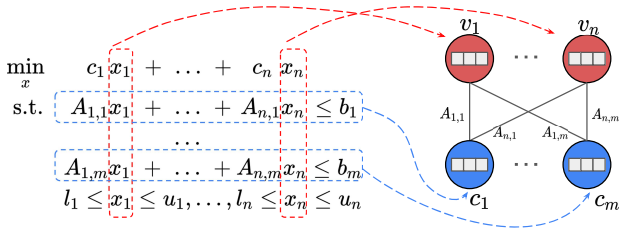


Fig. 1. The bipartite graph representation of an MILP, where node v_i and v_{n+j} indicates the i-th variable and the j-th constraint, respectively.

Graph Neural Networks For a graph $G = (\mathcal{V}, \mathcal{E})$, let $\mathcal{N}(v)$ denote the set of neighbors of v. The k-th message passing layer updates embeddings for each node v using the following formula:

$$h_v^{(k)} = f_2^{(k)} \left(\left\{ h_v^{(k-1)}, f_1^{(k)} \left(\left\{ h_u^{(k-1)} : u \in \mathcal{N}(v) \right\} \right) \right\} \right),$$

where $h_v^{(k)} \in \mathbb{R}^d$ denotes the hidden feature vector of node v in the k-th layer with $h_v^{(0)}$ being the initial embedding. Function $f_1^{(k)}(\cdot)$ is the *AGGREGATE* operator that gathers information from neighbors while function $f_2^{(k)}(\cdot)$ is the *COMBINE* operator that updates the aggregated representation. These two operators can take various choices, resulting in different architectures such as *graph convolutional networks* [14] and *graph attention networks* (GATs) [26].

Simplified Graph Transformers Transformers with global attention [25] can be considered a generalization of message passing to a fully connected graph. Typically, [28] proposed a simplified graph transformer that incorporates GNNs with a *linear attention function* defined as follows:

$$\mathbf{Q} = f_Q(\mathbf{H}^{(0)}), \quad \tilde{\mathbf{Q}} = \frac{\mathbf{Q}}{\|\mathbf{Q}\|_{\mathcal{F}}}, \quad \mathbf{K} = f_K(\mathbf{H}^{(0)}), \quad \tilde{\mathbf{K}} = \frac{\mathbf{K}}{\|\mathbf{K}\|_{\mathcal{F}}}, \quad \mathbf{V} = f_V(\mathbf{H}^{(0)}), \tag{3}$$

$$\mathbf{D} = \operatorname{diag}\left(\mathbf{1} + \frac{1}{N}\tilde{\mathbf{Q}}(\tilde{\mathbf{K}}^\top \mathbf{1})\right), \quad \mathbf{H} = \beta \mathbf{D}^{-1}\left[\mathbf{V} + \frac{1}{N}\tilde{\mathbf{Q}}(\tilde{\mathbf{K}}^\top \mathbf{V})\right] + (1-\beta)\mathbf{H}^{(0)} \tag{4}$$

where $\mathbf{H}^{(0)} \in \mathbb{R}^{|\mathcal{V}| \times d}$ represents the initial node embeddings, and f_Q, f_K, f_V denote shallow neural layers. $\mathbf{H}$ is the output of the attention module with β serving as a hyper-parameter for residual link. Given the simplified attention module, [28] utilized GNNs to incorporate structural information by adding the outputs of the two modules: $\mathbf{H}_O = (1-\alpha)\mathbf{H} + \alpha \mathrm{GNN}(\mathbf{H}^{(0)})$.

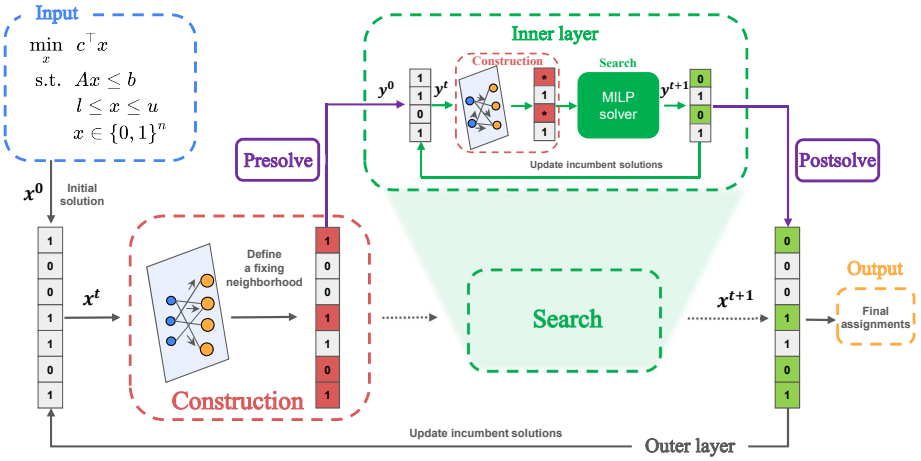


Fig. 2. An overview of our proposed learning-enhanced TLNS framework. The red part represents the learning-enhanced neighborhood construction stage while the green part denotes the neighborhood search stage. The purple part indicates that the presolve operator transforms the original MILP to a reduced MILP while the postsolve operator reverses the transformation.

3 The Learning-Based Two-Layer Large Neighborhood Search

In this section, we will provide a detailed description of the TLNS method and discuss how machine learning techniques contribute to fixing neighborhoods in TLNS. The overall algorithm is outlined in Fig. 2.

3.1 A TLNS Framework

Algorithm 1 Large Neighborhood Search (LNS)

Input: an MILP M, initial solution $\bar{x}$, fixing heuristic $\mathcal{D}$, count limit C, neighborhood size k, and adaptive rate η.
$cnt \leftarrow 0$
repeat
  $\mathcal{F} \leftarrow \mathcal{D}(M, \bar{x}, k)$
  solve $\mathcal{A}(M, \bar{x}, \mathcal{F})$ exactly and let x^* denote the corresponding solution
  if $c^\top x^* < c^\top \bar{x}$ **then**
    $\bar{x} \leftarrow x^*$
  else
    $k \leftarrow \eta \cdot k$
    $cnt \leftarrow cnt + 1$
  end if
until $cnt = C$ or time limit is reached
return $\bar{x}$

Algorithm 2 Two-Layer Large Neighborhood Search (TLNS)

Input: an MILP M, initial solution $\bar{x}$, fixing heuristic $\mathcal{D}$, presolve operator $\mathcal{Q}$, count limit C, neighborhood sizes k_1, k_2, and adaptive rates η_1, η_2.
repeat
  $\mathcal{F} \leftarrow \mathcal{D}(M, \bar{x}, k_1)$
  $\{P, \bar{y}\} \leftarrow \mathcal{Q}(\mathcal{A}(M, \bar{x}, \mathcal{F}), \bar{x})$ #presolve
  $y^* \leftarrow \text{LNS}(P, \bar{y}, \mathcal{D}, C, k_2, \eta_2)$
  $\{M, x^*\} \leftarrow \mathcal{Q}^{-1}(P, y^*)$ #postsolve
  if $c^\top x^* < c^\top \bar{x}$ **then**
    $\bar{x} \leftarrow x^*$
  else
    $k_1 \leftarrow \eta_1 \cdot k_1$
  end if
until time limit is reached
return $\bar{x}$

The classic LNS method starts with an initial feasible solution and then gradually refines it by iteratively constructing a fixing neighborhood and optimizing the corresponding auxiliary problem. The motivation for TLNS stems from the observation that auxiliary problems are still MILPs and hence can be further handled via LNS, rather than an off-the-shelf solver. We outline the classic LNS in Algorithm 1.

The Two-Layer Algorithm The TLNS framework can then be partitioned into two layers: an *outer layer* and an *inner layer*. Given an MILP (denoted by M) and an initial solution $\bar{x}$, we apply LNS iteratively to improve the incumbent solution, and this is called "outer layer". At each iteration, an auxiliary MILP has to be solved. Again, we apply LNS to it, which we call "inner layer". The pseudocode of TLNS is provided in Algorithm 2.

Outer Layer. Given an MILP M and its initial solution $\bar{x}$, one can apply a fixing heuristic to build a neighborhood of size k_1, defining an auxiliary problem $\mathcal{A}(M, \bar{x}, \mathcal{F})$. Before we directly call LNS to solve this auxiliary problem, we

need to deploy a critical operation: *presolve*. As [1] pointed out, presolve can be viewed as a collection of preprocessing techniques that reduce the size of and, more importantly, improve the "strength" of the given MILP, that is, the degree to which the constraints of the formulation accurately describe the underlying polyhedron of integer-feasible solutions. TLNS relies on a presolve operator $\mathcal{Q}$ to transform M and the incumbent $\bar{x}$ into a reduced problem P and $\bar{y}$, respectively. The incumbent $\bar{x}$ will be improved from the inner layer and the neighborhood size k_1 increases by a factor of $\eta_1 > 1$ if no improvements are made. The outer layer will terminate if the time limit is reached.

Inner Layer. The inner layer receives the presolved problem P along with its feasible solution $\bar{y}$ from the outer layer. A classic LNS is then invoked to optimize P. Specifically, we employ a count limit as the stopping criterion for the inner layer LNS, as described in Algorithm 1. During each iteration, if the inner layer fails to find a better solution to P within the neighborhood i.e. it is stuck in local minima, the neighborhood size k_2 should be increased by a factor of $\eta_2 > 1$ to facilitate exploration of a broader search space. Simultaneously, *cnt* is incremented to keep track of the number of times the neighborhood size has been augmented. In the end, a high-quality solution y^* to P is fed back to the outer layer where $\mathcal{Q}^{-1}$ transforms y^* back to its counterpart x^* in M.

Comparison between LNS and TLNS Let LNS($\mathcal{F}$) denote a single LNS process with the fixing neighborhood defined by $\mathcal{F}$ (i,e. optimizing $\mathcal{A}(M, \bar{x}, \mathcal{F})$). For convenience, let LNS($\mathcal{F}^1 : \mathcal{F}^H$) denote a process of applying LNS($\mathcal{F}^1$), LNS($\mathcal{F}^2$), ..., LNS($\mathcal{F}^H$) consecutively, with the superscript denoting its sequence. We utilize the subscript "1" and "2" to denote the outer and inner layer, respectively. Let TLNS($\mathcal{F}_1, \mathcal{F}_2^1 : \mathcal{F}_2^H$) denote a process of first enforcing the fixing neighborhood $\mathcal{B}(\mathcal{F}_1, \bar{x})$ in the outer layer and then applying LNS($\mathcal{F}_2^1 : \mathcal{F}_2^H$) in the inner layer.

Remark 1. Compared to LNS($\mathcal{F}_1$), TLNS($\mathcal{F}_1, \mathcal{F}_2^1 : \mathcal{F}_2^H$) exits search around the fixing neighborhood defined by $\mathcal{F}_1$ faster.

Both methods result in the same auxiliary problem $\mathcal{A}(M, \bar{x}, \mathcal{F}_1)$. The difference is that LNS($\mathcal{F}_1$) optimizes this problem via exact solvers while TLNS($\mathcal{F}_1, \mathcal{F}_2^1 : \mathcal{F}_2^H$) addresses such an MILP via LNS($\mathcal{F}_2^1 : \mathcal{F}_2^H$), a fast heuristic. Firstly, exact solvers are based on the branch-and-bound framework enhanced with a dozen of modules that are critical to exactness but computationally expensive, such as cutting-planes, domain propagation and symmetry-breaking [27]. Alternatively, LNS($\mathcal{F}_2^1 : \mathcal{F}_2^H$) channels all attention to searching for high-quality feasible solutions and is thus more efficient. Secondly, TLNS($\mathcal{F}_1, \mathcal{F}_2^1 : \mathcal{F}_2^H$) quickly identifies near-optimal solutions to $\mathcal{A}(M, \bar{x}, \mathcal{F}_1)$ and then exits search around the fixing neighborhood $\mathcal{B}(\mathcal{F}_1, \bar{x})$, moving towards a new neighborhood with potentially better solutions.

Remark 2. Compared to LNS($\mathcal{F}_1 \cup \mathcal{F}_2^1 : \mathcal{F}_1 \cup \mathcal{F}_2^H$), TLNS($\mathcal{F}_1, \mathcal{F}_2^1 : \mathcal{F}_2^H$) saves presolving time.

During step h, both methods utilize general-purpose solvers to optimize an auxiliary problem associated with $\mathcal{F}_1 \cup \mathcal{F}_2^h$. For LNS($\mathcal{F}_1 \cup \mathcal{F}_2^1 : \mathcal{F}_1 \cup \mathcal{F}_2^H$), the auxiliary problem is given as follows:

$$\min_{x \in S \cap \mathcal{B}\left(\mathcal{F}_1 \cup \mathcal{F}_2^h, \bar{x}\right)} \quad c^\top x. \tag{5}$$

Note that model (5) is the same as model (1) except that some variable bounds are fixed. As a result, presolving such models would become computationally costly, sometimes even exceeding its subsequent branch-and-bound tree search. Let T_p^1 and T_o denote the presolve time and the branch-and-bound search time, respectively. Then the total used time for LNS($\mathcal{F}_1 \cup \mathcal{F}_2^1 : \mathcal{F}_1 \cup \mathcal{F}_2^H$) is $H \times (T_p^1 + T_o)$. In TLNS($\mathcal{F}_1, \mathcal{F}_2^1 : \mathcal{F}_2^H$), the auxiliary problem $\mathcal{A}(M, \bar{x}, \mathcal{F}_1)$ in the outer layer is presolved only once, producing an MILP of a reduced size (denoted by P). Then an exact solver will be employed to first presolve $\mathcal{A}(P, \bar{y}, \mathcal{F}_2^h)$ (time T_p^2) with the subsequent branch-and-bound search (time T_o). The total used time for TLNS($\mathcal{F}_1, \mathcal{F}_2^1 : \mathcal{F}_2^H$) is $T_p^1 + H \times (T_p^2 + T_o)$. Given that T_p^2 is much smaller than T_p^1, the saved time is $(H-1) \times T_p^1 - H \times T_p^2$ when adopting TLNS($\mathcal{F}_1, \mathcal{F}_2^1 : \mathcal{F}_2^H$).

The above two remarks elucidate the advantage of TLNS over LNS. Typically, Remark 2 distinguishes between TLNS and LNS with smaller neighborhoods, highlighting the superiority of adopting such a nested approach over solely utilizing single-layer LNS with a small neighborhood.

3.2 Learning-Enhanced TLNS

We now utilize ML as a fixing heuristic in Algorithm 2. Let s^t denote the state of an MILP M with the incumbent solution x^t in step t. Our goal is to learn a policy $\pi_\theta(\cdot)$ that takes s^t as the input and returns scores to determine the fixing neighborhood. In the following, we first describe our training task and introduce the policy network, then we explain how we apply the learned policy for inference.

Training Following previous works [12,23], we employ LB as the expert and collect samples for training.

Definition 3 (LB neighborhood). *Consider an MILP with n variables, let $\bar{x}$ denote a feasible solution and k denote a distance cutoff parameter. Then an LB neighborhood is restricted to a ball around $\bar{x}$:*

$$\mathcal{B}(k, \bar{x}) \coloneqq \{x \in \mathbb{R}^n : \|x - \bar{x}\|_1 \leq k\},$$

where $\|\cdot\|_1$ denotes ℓ_1-norm.

Using an LB neighborhood, we define a sub-MILP (6).

$$\min_{x \in S \cap \mathcal{B}(k, \bar{x})} \quad c^\top x. \tag{6}$$

Model (6) is optimized by an MILP solver and let x^* denote its optimal solution. Since $x^* \in S$ and $c^\top x^* \leq c^\top \bar{x}$, x^* becomes a new incumbent solution. Let a binary vector a^* denote an action that can transform the previous incumbent

$\bar{x}$ to the new one x^*, i.e., $a_i^* \coloneqq |\bar{x}_i - x_i^*|$. We repeat the procedure of defining an LB neighborhood around the incumbent and optimizing the corresponding sub-MILP, until no objective improvement is achieved.

While optimizing a sub-MILP (6) in each iteration, we retrieve intermediate solution $\tilde{x}$ from the solution pool of an solver if $c^\top \bar{x} - c^\top \tilde{x} \geq \kappa_p(c^\top \bar{x} - c^\top x^*)$, with $0 < \kappa_p < 1$. These solutions are not necessarily optimal but of high quality, defining a *positive sample* set. Specifically, let $\mathcal{S}_p$ denote such a set consisting of action vectors that can transform $\bar{x}$ to $\tilde{x}$. We randomly flip elements of a^* by 10% while keeping the number of non-zero elements in a^* unchanged, which generates a new action a'. We then apply a' to $\bar{x}$ and induce an MILP of form (2) with x' being the optimal solution. We accept action a' as a *negative sample* if $c^\top \bar{x} - c^\top x' \leq \kappa_n(c^\top \bar{x} - c^\top x^*)$ with $0 < \kappa_n \leq \kappa_p$. Let $\mathcal{S}_n$ denote the set of negative samples. Finally, let $\mathcal{D} \coloneqq \{(s, \mathcal{S}_p, \mathcal{S}_n)\}$ denote the set of collected data.

We utilize *contrastive loss* for training. Formally, the loss function can be formulated as follows:

$$L(\theta) \coloneqq \sum_{(s,\mathcal{S}_p,\mathcal{S}_n)\in\mathcal{D}} \frac{-1}{|\mathcal{S}_p|} \sum_{a\in\mathcal{S}_p} \log \frac{\exp(a^\top \pi_\theta(s)/\tau)}{\sum\limits_{a'\in\mathcal{S}_n\cup\{a\}} \exp(a'^\top \pi_\theta(s)/\tau)},$$

where τ is the temperature hyper-parameter. The contrastive loss is deployed to bring $\pi_\theta(\cdot)$ closer to positive samples while simultaneously pushing it away from negative samples. When $|\mathcal{S}_p| = 1$ and $|\mathcal{S}_n| = 0$ (e.g., only optimal solutions are kept as samples), we reduce the contrastive loss to the classic cross entropy loss used in [23].

Policy Network The input of the policy is s^t and the output $\pi_\theta(s^t) \in [0,1]^n$ assigns scores for each variable. To encode s^t based on bipartite graph representations, previous works [12,23] adopted a rich set of features including information derived from solving linear programs (LPs). However, given the computational demands of these LP-based features in large-scale problems, we chose to include only those features that can be efficiently computed, as is deployed in [9]. Details are described in A.

Regarding the network architecture, while GNNs are naturally suited for bipartite graphs, the limited depth restricts interactions between distant variable nodes. Besides, the classic self-attention [25] layer with $O(n^2)$ complexity could be computationally prohibitive for large-scale MILPs. To address these limitations, we employ the Simplified Graph Transformer [28], which expands the GNNs' receptive field through a global attention module while maintaining a lightweight structure due to its linear attention mechanism. Specifically, we first employ the attention module described in model (4) to aggregate information across the entire graph. The output then serves as the initial node embedding of the subsequent GNN module, where we incorporate two interleaved half-convolution layers [7]. Finally, the embeddings of variable nodes are transformed into scalars within $[0,1]$ through 2-layer perceptrons alongside a sigmoid function. Please refer to Appendix B for details.

Table 1. Average size of each benchmark instance, the SMALL instances are used for data collection and training and the LARGE instances are used for testing

	SMALL				LARGE			
	SC	CA	MIS	MVC	SC	CA	MIS	MVC
# variables	4,000	4,000	6,000	1,000	16,000	100,000	100,000	20,000
# constraints	5,000	2,662	15,157	65,100	20,000	794,555	5,001,669	3,960,000
# non-zeros	1,000,000	22,757	30,314	130,200	16,000,000	4,000,000	10,003,338	7,920,000

Inference In LNS, when building a fixing neighborhood of size r, we apply the learned policy $\pi_\theta(\cdot)$ to inform the neighborhood. We employ a sampling strategy to randomly select r variables to be unfixed without replacement according to $\pi_\theta(s)$, where variables with higher scores are more likely to be selected.

4 Numerical Experiments

In this section, we evaluate the performance of our proposed algorithm and compare it with other methods. The code will be made publicly available upon publication. Our code is available at https://github.com/NetSysOpt/TLNS

4.1 Setup

Benchmarks In our evaluation, we assess our algorithm on four widely used $\mathcal{NP}$-hard problem benchmarks—Set Cover (SC), Combinatorial Auction (CA), Maximum Independent Set (MIS), and Minimum Vertex Cover (MVC)—following the instance generation procedures of [12]. Each SC instance is generated with $5,000$ items and $4,000$ subsets, while every CA instance is generated with $4,000$ bids and $2,000$ items. MIS and MVC are graph-related problems and they are generated from random graphs with $6,000$ and $1,000$ nodes, average degrees of 5 and 130, respectively. For each benchmark, we create $1,000$ instances (denoted by SMALL), split into training and validation sets of 900 and 100 instances, respectively. Additionally, we generate 20 larger instances (denoted by LARGE) for each benchmark, split in half for test and validation sets. LARGE instances range from 4 to 25 times the size of SMALL instances with MILP sizes specified in Table 1 (refer to Appendix C for more details). These MILPs have up to 100 thousand variables and 5 million constraints, becoming computationally prohibitive for general-purpose solvers as we will show in Section 4.4.

Evaluation Configurations All evaluations are performed under the same configuration. The evaluation machines include 12th Gen Intel(R) Core(TM) i9-12900K CPUs with Nvidia GeForce RTX 3090 GPUs. For off-the-shelf MILP solvers, Gurobi 10.0.2 [8] and SCIP 8.0.4 [3] are utilized in our experiments. The time limit for running each experiment is set to $1,000$ seconds since a tail-off of solution qualities was often observed after that.

Data collection & Training For each training instance, we utilize Gurobi with a solution limit of 1 to generate the very first incumbent solution. Using this incumbent, we apply the LB heuristic with a fine-tuned neighborhood size of 100, 400, 500, and 75 for SC, CA, MIS, and MVC, respectively. The resulting LB-MILP is optimized by Gurobi with a time limit of $1,500$ seconds. We employ the contrastive loss with τ being equal to 0.07. The batch size is set to 32 and Adam [13] with a learning rate of 0.001 is utilized as the optimizer. We remark that in our experiments: (i) The Graph Transformer models are trained using SMALL instances but applied to LARGE ones; (ii) the same models are deployed in LNS as well as both layers of TLNS.

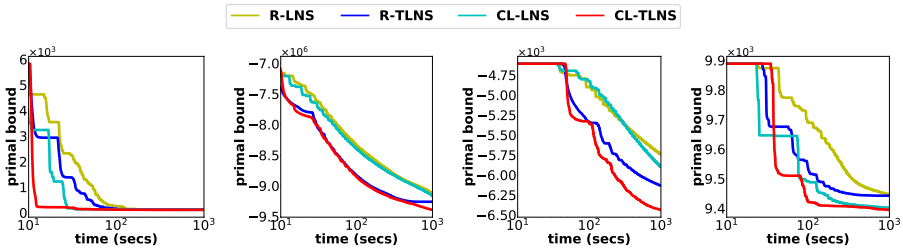
Metrics In order to assess the performance of different methods, we employ two metrics: (i) *primal bound* (PB), which refers to the objective value; (ii) *primal integral* (PI), which measures the integral of primal gap with respect to runtime, where the primal gap denotes a normalized difference between the primal bound and a pre-specified best known objective value. The best known objectives correspond to the best solutions returned by all methods evaluated in our experiments with a longer time limit of $3,600$ seconds. The PB value demonstrates the quality of returned solutions at runtime while the PI value provides insight into the speed of identifying better solutions. Note that we choose not to report the primal gap as a measure of optimality in our experiments since for large-sized instances, both Gurobi and SCIP could neither identify optimal/near-optimal solutions nor provide tight dual bounds with a reasonable time limit (e.g. $500,000$ seconds).

Since all four benchmarks considered in our experiments entail minimization problems, smaller PB and PI values imply better computational performances.

To showcase the effectiveness of TLNS, we conduct the following progressive experiments: (i) comparing TLNS with LNS under classic settings (Section 4.2); (ii) comparing TLNS with LNS under learning-based settings (Section 4.3); and (iii) comparing against state-of-the-art MILP solvers (Section 4.4). Readers are referred to Appendix E for additional experiments.

4.2 Comparison between TLNS and LNS (classic)

We hypothesize that our proposed TLNS outperforms LNS irrespective of neighborhood-fixing heuristics. To demonstrate this, we first compare TLNS against LNS under classic learning-free settings. Among various heuristics for fixing variables, we employ a *random* heuristic, which has been used in [20] and is straightforward to implement as it selects variables randomly. We denote LNS and TLNS with the random heuristic by `R-LNS` and `R-TLNS`, respectively. The very first incumbent solution is again provided by Gurobi with a solution limit of one. We employ SCIP as the off-the-shelf MILP solver in both `R-LNS` and `R-TLNS` since the presolve operator in Gurobi is inaccessible. The neighborhood sizes of both methods are fine-tuned separately and details of those hyper-parameters are provided in Appendix D.



(a) SC (b) CA (c) MIS (d) MVC

Fig. 3. The PB, as a function of runtime, averaged over 10 instances. Lower PB values imply better performance.

Fig 3 depicts the average PB as a function of runtime. Clearly, the PB value of `R-TLNS` (blue) is smaller than that of `R-LNS` (yellow) for most of the runtime, demonstrating that our two-layer neighborhood search method indeed exceeds the single-layer version. Table 2 presents PI and PB values at a time limit of 1,000 seconds, averaged over 10 instances for each dataset, along with their respective standard deviations. Bold values in columns "`R-LNS`" and "`R-TLNS`" indicate superior performances. Column "Gain" presents the improvement of `R-TLNS` over `R-LNS` for each metric. For instance, in the case of MVC, the PI values for `R-LNS` and `R-TLNS` are 14.5 and 9.0 respectively. The gain is computed as $(14.5 - 9.0)/14.5 = 37.9\%$. Notably, in terms of PI, `R-TLNS` surpasses `R-LNS` across all datasets, particularly exhibiting a remarkable improvement of 51.3% on the CA dataset. This indicates that `R-TLNS` produces high-quality solutions faster than `R-LNS`. In terms of PB at 1,000 seconds, `R-TLNS` produces better solutions to CA and MIS, equivalently good ones to MVC, and slightly worse ones to SC, compared with `R-LNS`. The latter phenomenon is potentially due to the fact that both methods stagnate in the later runtime but `R-LNS` relying on exact solvers is capable of producing optimal solutions to auxiliary problems. We claim that `R-TLNS` generally outperforms `R-LNS`.

Table 2. PI and PB values at 1,000 seconds for `R-LNS`, `R-TLNS`, `CL-LNS` `CL-TLNS`, SCIP and Gurobi averaged over 10 instances for each benchmark, along with their standard deviations. Lower PI/PB values imply better performances.

Dataset		`R-LNS`	`R-TLNS`	Gain	`CL-LNS`	`CL-TLNS`	Gain	SCIP	Gurobi	Gain
SC	PI	1695.7±178.7	**1250.9±137.8**	**26.2%**	871.8±58.5	**633.5± 43.6**	**27.3%**	262487±7482	16294±1035	96.1%
	PB	**118.0±3.8**	121.3±6.17	-2.7%	**112.7±2.7**	113.0±2.9	-0.2%	116.6±2.6	120.9±2.7	3.0%
CA	PI	64.2±3.4	**31.2±4.3**	**51.3%**	60.5±2.3	**25.5±1.7**	**57.8%**	285.3±5.7	180.3±3.4	85.8%
	PB	-9124241.2±29637.7	**-9260001.4±34248.5**	**1.4%**	-9162882.8±38388.0	**-9391997.4±50556.4**	**2.5%**	-7071993.8±50583.6	-7728867.4±36376.9	21.5%
MIS	PI	158.5±4.5	**92.8±2.1**	**41.4%**	149.4±17.0	**50.6± 4.9**	**66.1%**	–	284.9±3.9	82.2%
	PB	-5740.5±22.7	**-6134.7±17.6**	**6.8%**	-5897.8±85.6	**-6435.8±16.8**	**9.1%**	–	-4606.7±20.2	39.7%
MVC	PI	14.5±0.8	**9.0±0.7**	**37.9%**	5.1±0.6	**3.7± 0.3**	**26.5%**	45.3±0.8	41.5±3.3	90.9%
	PB	9446.6±39.9	**9442.7± 42.5**	**0.04%**	9401.8±39.6	**9394.8±41.5**	**0.07%**	9602.8±42.4	9751.1±50.2	2.1%

4.3 Comparison between TLNS and LNS (learning)

We proceed to show that TLNS performs better than LNS in learning-based settings. The very recent effort of enhancing LNS with ML is the work of [12], where the authors adopted contrastive learning to build fixing neighborhoods. We apply this technique in the TLNS algorithm, denoting the two methods as `CL-LNS` and `CL-TLNS`, respectively. `SCIP` is used as the off-the-shelf MILP solver within both methods. `CL-LNS` and `CL-TLNS` are then evaluated on four benchmark datasets. The neighborhood sizes are tuned separately for each method and their details are provided in Appendix D. To ensure fairness, the same trained models are utilized in both methods.

We plot PB as a function of runtime in Fig 3. The PB curves of `CL-TLNS` (red) are almost consistently below that of `CL-LNS` (cyan). Table 2 presents the PI/PB metrics of both methods at the time limit of $1,000$ seconds. In terms of PI, `CL-TLNS` significantly surpasses `CL-LNS` across all four benchmark datasets–with an improvement ranging from 26.5% to 66.1%. In terms of PB, `CL-TLNS` produces better solutions to CA and MIS, equivalently good ones to MVC, and slightly worse ones to SC, compared with `CL-LNS`. We found out that MVC instances were solved to near-optimality, hence `CL-TLNS` and `CL-LNS` achieve comparable PB metrics. As for SC, the mildly worse performance of `CL-TLNS` can be again attributed to stagnation, as discussed in Section 4.2. We can claim that `CL-TLNS` generally outperforms `CL-LNS`.

We now compare the learning-guided fixing method with a random heuristic used in TLNS. From Fig 3, the PB value of `CL-TLNS` (red) is almost consistently lower than that of `R-TLNS` (blue) on all four benchmark datasets. Notably, switching from a random heuristic to a learning-based one benefits TLNS significantly across all datasets. From Table 2, we compare both PB and PI metrics of `CL-TLNS` with those of `R-TLNS`. Again, we can claim that `CL-TLNS` generally performs better than `R-TLNS`.

4.4 Comparison against MILP Solvers

We compare `CL-TLNS` against Gurobi and SCIP across four datasets. To ensure a fair comparison between exact solvers and heuristics, we enforce both solvers to apply their internal heuristics more aggressively. In particular, we use "model.setHeuristics(SCIP_PARAMSETTING.AGGRESSIVE)" for SCIP and set the parameter "MIPFocus = 1" for Gurobi. Table 2 exhibits the averaged PI and PB metrics. Note that "–" in Column "SCIP" indicates that SCIP is incapable of handling MIS problems due to memory limits. The underlined values in columns "SCIP" and "Gurobi" signify better performances between these two solvers while column "Gain" represents the improvement of `CL-TLNS` over the superior solver in terms of respective metrics. The computational results show that `CL-TLNS` consistently outperforms Gurobi and SCIP across all benchmarks, achieving an improvement of up to 96.1% and 39.7% in PI and PB, respectively.

5 Conclusions

This paper proposes a learning-enhanced TLNS method especially for addressing large-scale MILPs. Classic LNS methods refine incumbent solutions by building a particular neighborhood and searching within such a region by optimizing auxiliary MILPs via off-the-shelf solvers while our proposed TLNS goes one step further and solves auxiliary problems via LNS. Graph transformer models are incorporated into TLNS for guiding neighborhood construction, boosting the performance of TLNS. We argue that learning-based TLNS would outperform classic LNS and demonstrate this in our experiments. The results show that TLNS achieved significantly better performances in identifying high-quality solutions within a short time frame. An immediate research direction is to generalize TLNS and to extend it to the *multi-layer* LNS, where LNS is applied recursively to solve subsequent sub-MILPs. The *multi-layer* LNS is promising to address extremely large-scale MILPs while reducing reliance on off-the-shelf solvers.

Acknowledgment. This research is supported by the National Key R&D Program of China (Grant No. 2022YFA1003900), National Natural Science Foundation of China (Grant No. 12301416), Guangdong Basic and Applied Basic Research Foundation (Grant No. 2024A1515010306), Hetao Shenzhen-Hong Kong Science and Technology Innovation Cooperation Zone Project (No. HZQSWS-KCCYB-2024016), and Longgang District Special Funds for Science and Technology Innovation (Grant No. LGKCSDPT2023002).

References

1. Achterberg, T., Bixby, R.E., Gu, Z., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. INFORMS Journal on Computing **32**(2), 473–506 (2020)
2. Berthold, T.: Rens: the optimal rounding. Mathematical Programming Computation **6**, 33–54 (2014)
3. 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., Witzig, J.: The SCIP Optimization Suite 8.0. ZIB-Report 21-41, Zuse Institute Berlin (December 2021), `http://nbn-resolving.de/urn:nbn:de:0297-zib-85309`
4. Danna, E., Rothberg, E., Pape, C.L.: Exploring relaxation induced neighborhoods to improve mip solutions. Mathematical Programming **102**, 71–90 (2005)
5. Fischetti, M., Lodi, A.: Local branching. Mathematical programming **98**, 23–47 (2003)
6. Gasse, M., Bowly, S., Cappart, Q., Charfreitag, J., Charlin, L., Chételat, D., Chmiela, A., Dumouchelle, J., Gleixner, A., Kazachkov, A.M., et al.: The machine learning for combinatorial optimization competition (ml4co): Results and insights. In: NeurIPS 2021 Competitions and Demonstrations Track. pp. 220–231. PMLR (2022)

7. Gasse, M., Chételat, D., Ferroni, N., Charlin, L., Lodi, A.: Exact combinatorial optimization with graph convolutional neural networks. Advances in neural information processing systems **32** (2019)
8. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2023), `https://www.gurobi.com`
9. Han, Q., Yang, L., Chen, Q., Zhou, X., Zhang, D., Wang, A., Sun, R., Luo, X.: A gnn-guided predict-and-search framework for mixed-integer linear programming. arXiv preprint arXiv:2302.05636 (2023)
10. Hendel, G.: Adaptive large neighborhood search for mixed integer programming. Mathematical Programming Computation pp. 1–37 (2022)
11. Huang, T., Ferber, A., Tian, Y., Dilkina, B., Steiner, B.: Local branching relaxation heuristics for integer linear programs. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 96–113. Springer (2023)
12. Huang, T., Ferber, A.M., Tian, Y., Dilkina, B., Steiner, B.: Searching large neighborhoods for integer linear programs with contrastive learning. In: International Conference on Machine Learning. pp. 13869–13890. PMLR (2023)
13. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
14. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
15. Li, Q., Han, Z., Wu, X.M.: Deeper insights into graph convolutional networks for semi-supervised learning. In: Proceedings of the AAAI conference on artificial intelligence. vol. 32 (2018)
16. Liu, D., Fischetti, M., Lodi, A.: Learning to search in local branching. In: Proceedings of the aaai conference on artificial intelligence. vol. 36, pp. 3796–3803 (2022)
17. Luathep, P., Sumalee, A., Lam, W.H., Li, Z.C., Lo, H.K.: Global optimization method for mixed transportation network design problem: a mixed-integer linear programming approach. Transportation Research Part B: Methodological **45**(5), 808–827 (2011)
18. Nair, V., Bartunov, S., Gimeno, F., Von Glehn, I., Lichocki, P., Lobov, I., O'Donoghue, B., Sonnerat, N., Tjandraatmadja, C., Wang, P., et al.: Solving mixed integer programs using neural networks. arXiv preprint arXiv:2012.13349 (2020)
19. Pochet, Y., Wolsey, L.A.: Production planning by mixed integer programming, vol. 149. Springer (2006)
20. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS Journal on Computing **19**(4), 534–541 (2007)
21. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: International conference on principles and practice of constraint programming. pp. 417–431. Springer (1998)
22. Song, J., Yue, Y., Dilkina, B., et al.: A general large neighborhood search framework for solving integer linear programs. Advances in Neural Information Processing Systems **33**, 20012–20023 (2020)
23. Sonnerat, N., Wang, P., Ktena, I., Bartunov, S., Nair, V.: Learning a large neighborhood search algorithm for mixed integer programs. CoRR **abs/2107.10201** (2021), `https://arxiv.org/abs/2107.10201`
24. Toth, P., Vigo, D.: The vehicle routing problem. SIAM (2002)
25. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)

26. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
27. Wolsey, L.A., Nemhauser, G.L.: Integer and combinatorial optimization, vol. 55. John Wiley & Sons (1999)
28. Wu, Q., Zhao, W., Yang, C., Zhang, H., Nie, F., Jiang, H., Bian, Y., Yan, J.: Simplifying and empowering transformers for large-graph representations. Advances in Neural Information Processing Systems **36** (2024)
29. Wu, Y., Song, W., Cao, Z., Zhang, J.: Learning large neighborhood search policy for integer programming. Advances in Neural Information Processing Systems **34**, 30075–30087 (2021)
30. Zhang, J., Liu, C., Li, X., Zhen, H.L., Yuan, M., Li, Y., Yan, J.: A survey for solving mixed integer programming via machine learning. Neurocomputing **519**, 205–217 (2023)

A Feature Selection

Following the work of [9], we adopt the feature selection strategy described in Table 3.

Table 3. Features selection in bipartite representations

	name	description
Variable	obj	normalized coefficient of variables in the objective function
	v_coeff	average coefficient of the variable in all constraints
	Nv_coeff	degree of variable node in the bipartite representation
	max_coeff	maximum value among all coefficients of the variable
	min_coeff	minimum value among all coefficients of the variable
	int	binary representation to show if the variable is an integer variable
	cur_sol	value in the current solution
Constraint	c_coeff	average of all coefficients in the constraint
	Nc_coeff	degree of constraint nodes in the bipartite representation
	rhs	right-hand-side value of the constraint
	sense	the sense of the constraint
	norm	ℓ_2-norm of the coefficient vector(e.g. the vector formed by all coefficients in the constraint.)
	cos_sim	cosine similarity between the coefficient vectors of the constraint and objective.
Edge	coeff	coefficient of variables in constraints

B Network Architecture

The input is an LNS state s^t and the output is $a \in [0,1]^n$ that assigns scores for each variable node. Initially, s^t is encoded based on the bipartite representation with features described in Table 3. All features are initially embedded into $\mathbb{R}^d$ through linear transformation and are denoted as $\mathbf{H}^{(0)} \in \mathbb{R}^{|\mathcal{V}|\times d}$. Then, the simplified attention module is employed.

$$\mathbf{Q} = f_Q(\mathbf{H}^{(0)}), \quad \tilde{\mathbf{Q}} = \frac{\mathbf{Q}}{\|\mathbf{Q}\|_{\mathcal{F}}}, \quad \mathbf{K} = f_K(\mathbf{H}^{(0)}), \quad \tilde{\mathbf{K}} = \frac{\mathbf{K}}{\|\mathbf{K}\|_{\mathcal{F}}}, \quad \mathbf{V} = f_V(\mathbf{H}^{(0)}),$$

$$\mathbf{D} = \operatorname{diag}\left(\mathbf{1} + \frac{1}{N}\tilde{\mathbf{Q}}(\tilde{\mathbf{K}}^\top \mathbf{1})\right), \quad \mathbf{H} = \beta \mathbf{D}^{-1}\left[V + \frac{1}{N}\tilde{\mathbf{Q}}(\tilde{\mathbf{K}}^\top \mathbf{V})\right] + (1-\beta)\mathbf{H}^{(0)}$$

where f_Q, f_K, f_V are shallow neural layers and β is a hyper-parameter for residual link. With $\mathbf{H} \in \mathbb{R}^{|\mathcal{V}|\times d}$ being the output of the attention module, we consider each row $\mathbf{H}_i$ of $\mathbf{H}$ as the initial embedding in the subsequent GNN module, e.g. $h_{v_i}^{(0)} = \mathbf{H}_i$. Then two interleaved half-convolutions [7] are employed.

$$h_{v_i}^{(k)} = f_c^{(k)}\left(h_{v_i}^{(k-1)}, \sum_{u\in\mathcal{N}(v_i)} h_u^{(k-1)}\right), \quad i \in \{n+1, n+2, ..., n+m\}$$

$$h_{v_i}^{(k)} = f_v^{(k)}\left(h_{v_i}^{(k-1)}, \sum_{u\in\mathcal{N}(v_i)} h_u^{(k)}\right), \quad i \in \{1, 2, ..., n\}$$

where $f_c^{(k)}, f_v^{(k)}$ are 2-layer perceptrons with ReLU activation functions. Finally, the embedding of each variable node is transformed into a scalar by a 2-layer perceptron and bounded between $[0, 1]$ by a sigmoid function.

C Benchmark Instances

1. **Set Cover**
   In the SC instance, we are given a ground set of m items and $\mathcal{S}$ is the collection of n subsets of $[m]$. The aim is to choose the minimal number of subsets from $\mathcal{S}$ such that the union of the selected sets cover all the elements in $[m]$:
$$\begin{aligned} \min_{x} \quad & \sum_{s\in\mathcal{S}} x_s \\ \text{s.t.} \quad & \sum_{s\in\mathcal{S}:i\in s} x_s \geq 1,\ \forall i \in [m], \\ & x_s \in \{0,1\},\ \forall s \in \mathcal{S} \end{aligned}$$
2. **Combinatorial Auction**
   In the CA instance, there are n bids $\{(p_i, B_i) : i \in [n]\}$ for m items. B_i is the subset of items of the i-th bid with p_i being the corresponding bidding price. The goal is to assign items to maximize the overall revenue.
$$\begin{aligned} \min_{x} \quad & -\sum_{i\in[n]} p_i x_i \\ \text{s.t.} \quad & \sum_{i:j\in B_i} x_i \leq 1,\ \forall j \in [m], \\ & x_i \in \{0,1\},\ \forall i \in [n] \end{aligned}$$
3. **Maximum Independent Set**
   In the MIS instance, we are given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The objective is to select the maximum number of nodes such that no two selected nodes are adjacent to each other.
$$\begin{aligned} \min_{x} \quad & -\sum_{v\in\mathcal{V}} x_v \\ \text{s.t.} \quad & x_u + x_v \leq 1,\ \forall (u,v) \in \mathcal{E}, \\ & x_v \in \{0,1\},\ \forall v \in \mathcal{V} \end{aligned}$$
4. **Minimum Vertex Cover**
   In the MVC instance, we are given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The goal is to select minimum number of nodes such that for every edge $(u, v) \in \mathcal{E}$, at least one endpoint of (u, v) is selected.
$$\begin{aligned} \min_{x} \quad & \sum_{v\in\mathcal{V}} x_v \\ \text{s.t.} \quad & x_u + x_v \geq 1,\ \forall (u,v) \in \mathcal{E} \\ & x_v \in \{0,1\},\ \forall v \in \mathcal{V} \end{aligned}$$

D Hyper-Parameters

Table 4 exhibits the neighborhood sizes that are employed in our experiments. Particularly, the neighborhood sizes (500/120, for example) of TLNS indicates that 500 is the number of variables unfixed in the outer layer while 120 is the number of variables unfixed in the inner layer. Other hyper-parameters are introduced in Table 5.

Table 4. Neighborhood sizes for different datasets

Dataset	R-LNS	R-TLNS	CL-LNS	CL-TLNS
SC	4,000	8,000/1600	175	500/120
CA	35,000	60,000/3,000	35,000	60,000/3,000
MIS	40,000	70,000/7,000	12,500	30,000/7,000
MVC	10,000	15,000/1,250	1,250	5,500/1,000

Table 5. Hyper-parameter configurations

hyper-parameter	notations	value
Suboptimality threshold to determine positive samples	κ_p	0.6
Suboptimality threshold to determine negative samples	κ_n	0.1
Temperature in contrastive loss	τ	0.07
Feature embedding dimension	d	32
Outer adaptive rate	η_1	1.05
Inner adaptive rate	η_2	1.15
Runtime limit for solvers in LB(data collection)	T_{LB}	$1,500$ seconds
Runtime limit for solvers in LNS	T_{sub}	50 seconds
Time limit	T	$1,000$ seconds
Count limit	C	6/3/4/4 for SC / CA / MIS / MVC

We utilize the performance on the validation set for hyper-parameter tuning across all methods, hyper-parameters are selected through grid search within the following search space:

- $T_{sub} \in \{25, 50, 100\}$
- $\eta_1 \in \{1.02, 1.05, 1.1\}$
- $\eta_2 \in \{1.05, 1.15, 1.25\}$

While the above hyper-parameters are not sensitive to certain methods or benchmarks, we adopt the ones that are generally suitable across all benchmarks. On the other hand, noting that the neighborhood sizes are crucial hyper-parameters in LNS-based methods, we tune the neighborhood sizes for each method across all benchmarks. The search space is described in Table 6

Table 6. Neighborhood sizes tuning

dataset	R-LNS	R-TLNS	CL-LNS	CL-TLNS
SC	$k_1 \in \{3000, 4000, 5000\}$	$k_1 \in \{6000, 8000, 10000\}$ $k_2 \in \{1400, 1600, 1800\}$	$k_1 \in \{100, 125, 150, 175, 200, 300\}$	$k_1 \in \{500, 800, 1000\}$ $k_2 \in \{80, 100, 120\}$
CA	$k_1 \in \{25000, 35000, 45000\}$	$k_1 \in \{50000, 60000, 70000\}$ $k_2 \in \{2000, 3000, 4000\}$	$k_1 \in \{25000, 30000, 35000, 40000, 45000\}$	$k_1 \in \{50000, 60000, 70000\}$ $k_2 \in \{2000, 3000, 4000\}$
MIS	$k_1 \in \{30000, 40000, 50000\}$	$k_1 \in \{60000, 70000, 80000\}$ $k_2 \in \{6000, 7000, 8000\}$	$k_1 \in \{10000, 12500, 15000, 20000, 30000\}$	$k_1 \in \{25000, 30000, 35000\}$ $k_2 \in \{5000, 6000, 7000\}$
MVC	$k_1 \in \{9000, 10000, 11000\}$	$k_1 \in \{12500, 15000, 17500\}$ $k_2 \in \{1250, 1500, 1750\}$	$k_1\{1000, 1250, 1500, 1750, 2000\}$	$k_1 \in \{5000, 5500, 6000\}$ $k_2 \in \{750, 1000, 1250\}$

E Additional Experiments

E.1 Comparison against LP-free Heuristics

Table 7. PI and PB at 1,000 seconds for `CL-TLNS`, `GRB(NoRel)` and `CL-LNS(NoRel)`, averaged over 10 instances for each benchmark, along with their standard deviations. Lower PI/PB values imply better performances.

Dataset		CL-TLNS	`GRB(NoRel)`	`CL-LNS(NoRel)`	Gain
SC	PI	**633.5±43.6**	15378.4±1028.7	2473.6±242.5	74.4%
	PB	113.0±2.9	114.9±2.7	**112.6±2.8**	-0.3%
CA	PI	**25.5±1.7**	41.3±4.9	55.9±4.6	38.2%
	PB	**-9391997.4±50556.4**	-9283749.8±57926.9	-9300736.2±33056.6	1.0%
MIS	PI	**50.6±4.9**	182.5±5.7	68.0±7.6	25.6%
	PB	**-6435.8±16.8**	-6009.2±28.4	-6298.4±17.7	2.1%
MVC	PI	**3.7±0.3**	16.2±0.3	3.7±0.4	0.0%
	PB	9394.8±41.5	9423.6±38.3	**9394.7±39.4**	0.0%

In Section 4.4, we compare our approach with advanced MILP solvers. However, for large-scale problems, these solvers can be inefficient due to their exact search nature and the high computational cost of solving LP relaxations, which do not necessarily contribute to improving PB values. To address these inefficiencies, the current section introduces an alternative baseline by employing LP-free heuristics. Specifically, we use Gurobi with "NoRel" (No Relaxation) heuristics to (i) directly solve MILPs (denoted as `GRB(NoRel)`) and (ii) serve as the underlying solver for `CL-LNS` (denoted as `CL-LNS(NoRel)`). We then compare `CL-TLNS` with both `GRB(NoRel)` and `CL-LNS(NoRel)`, with the results presented in Table 7. Bald values in the columns labeled "`CL-TLNS`", "`GRB(NoRel)`" and "`CL-LNS(NoRel)`" indicate the better performance between these three approaches. In terms of PI, `CL-TLNS` significantly outperforms the other baselines on SC, CA, and MIS, while achieving comparable results to `CL-LNS(NoRel)` on MVC. Regarding PB, `CL-TLNS` delivers the best solutions for CA and MIS but slightly underperforms `CL-LNS(NoRel)` on SC, which can be attributed to stagnation. Notably, although

`CL-LNS(NoRel)` relies on Gurobi as its underlying solver, `CL-TLNS` still outperforms it, despite using SCIP as its MILP solver. We conclude that `CL-TLNS` outperforms both `GRB(NoRel)` and `CL-LNS(NoRel)`.

It is also noteworthy that, among the three heuristics, the number of LNS layers in `GRB(NoRel)`, `CL-LNS(NoRel)`, and `CL-TLNS` are 0, 1, and 2, respectively. As the number of layers increases, performance improves. This indicates that adding more layers can be an effective strategy for tackling large-scale problems.

E.2 Ablation Study

In this section, ablation experiments are conducted to exhibit the performance of the different model architecture. Specifically, let SGT denote the simplified graph transformer model described in Appendix B. Let GCN denote the classic graph neural network with half-convolutions (e.g. SGT without attention layer) and let GAT denote the Graph Attention Network with half-convolutions deployed in [12]. The hidden dimensions for all models are set to 32.

Table 8. PI and PB at 1,000 for CL-TLNS with model architecture SGT, GCN, GAT. Lower PI/PB values imply better performances.

dataset		SGT	GCN	GAT
SC	PI	**633.5±43.6**	640.6±33.8	OOM
	PB	**113.0±2.9**	113.1±4.0	OOM
CA	PI	**25.5±1.7**	196.7±6.6	35.1±3.5
	PB	**-9391997.4± 50556.4**	-8035948.2±82793.1	-9358284.7±16573.2
MIS	PI	**50.6±4.9**	170.3±3.3	78.2±3.5
	PB	**-6435.8±16.8**	-5602.9±37.8	-6162.3±13.7
MVC	PI	3.7±0.3	5.25±0.5	**3.17±0.3**
	PB	9394.8±41.5	9401.1±42.2	**9394.1±41.8**

The results are exhibited in Table 8. On the one hand, our SGT model outperforms GCN, enhancing the performance of the model through expanded receptive fields. On the other hand, SGT outperforms GAT on CA and MIS, while achieving comparable performance on MVC. Notably, GAT is practically more memory-intensive and may encounter memory issues (e.g. out of memory (OOM) on SC), which SGT manages to avoid.

E.3 Ablation of Presolve

In Remark 2 we highlight that TLNS saves presolving time compared to LNS with small neighborhoods. Although the presolve procedure is often essential during optimization, it can be disabled. To strengthen Remark 2, we conduct experiments to evaluate the performance of LNS without presolve (i.e. the presolve procedure

is omitted when solving sub-MILPs). Specifically, we consider the following two baselines (i) CL-LNS without presolve (denoted as `CL-LNS(np)`) and (ii)CL-LNS with small neighborhoods and without presolve (denoted as `CL-LNS(np&sn)`). Both baselines use Gurobi with parameter Presolve=0 as the underlying solver whereas `CL-TLNS` employs SCIP.

Table 9. PI and PB at 1,000 seconds time limit for `CL-TLNS`, `CL-LNS(np)` and `CL-LNS(np&sn)`. Lower PI/PB values imply better performance

DATASET		`CL-TLNS`	`CL-LNS(np)`	`CL-LNS(np&sn)`
SC	PI	**633.5±43.6**	3469.4±627.8	734.8±79.7
	PB	113.0±2.9	**112.6±2.2**	113.0±2.9
CA	PI	**25.5±1.7**	39.3±5.3	108.2±4.8
	PB	**-9391997.4±50556.4**	-9265724.3±43317.0	-8925042.3±65739.2
MIS	PI	**50.6±4.9**	78.1±10.8	201.3±14.6
	PB	**-6435.8±16.8**	-6222.8±39.7	-5453.0±90.0
MVC	PI	**3.7±0.3**	6.0±0.6	5.2±0.6
	PB	**9394.8±41.5**	9400.0±40.7	9411.9±38.6

Table 9 exhibits the PI/PB values of the three approaches. From the results, we conclude that `CL-TLNS` consistently outperforms the other methods across most datasets. These findings highlight the importance of presolve, while TLNS demonstrates its superiority in reducing presolving time.

E.4 Experiments on small instances

The `CL-TLNS` is proposed especially for tackling large-scale MILPs. In this section, we explore the performance of `CL-TLNS` on small instances (e.g., instances of the same size as the training instances). The results are shown in Table 10, from which we can conclude that while TLNS is tailored for addressing large-scale MILPs, it still achieves comparable performance to the single-layer version LNS on small instances. It is important to note that our `CL-TLNS` significantly outperforms the state-of-the-art solvers even on small instances, with the exception of CA, where `CL-TLNS` is superior to SCIP but not as effective as Gurobi. This may be attributed to our use of SCIP as the underlying solver instead of Gurobi.

Table 10. PI and PB at 1,000 seconds time limit for `CL-LNS`, `CL-TLNS`, `SCIP` and `Gurobi` on small instances. Lower PI/PB values imply better performance

DATASET		`CL-LNS`	`CL-TLNS`	`SCIP`	`Gurobi`
SC	PI	**81.2±67.9**	82.6±69.0	2869.1±540.5	545.3±123.7
	PB	**170.5±17.1**	170.8±17.7	174.6±18.6	172.0±18.5
CA	PI	18.1±11.2	19.2±11.2	121.2±32.1	**17.7±10.8**
	PB	-114364.1±1748.9	-114377.7±1313.1	-112330.8±1274.8	**-114889.6±1367.2**
MIS	PI	**3.6±4.4**	3.7±4.5	31.8±8.5	99.4±12.3
	PB	-2623.3±12.3	**-2623.8±12.2**	-2574.8±20.5	-2368.4±33.9
MVC	PI	10.7±13.9	**10.4±12.9**	26.7±21.0	13.7±14.3
	PB	439.6±7.8	**439.2±7.7**	446.3±8.9	440.5±7.9