RL-MILP Solver: A Reinforcement Learning Approach for Solving Mixed-Integer Linear Programs with Graph Neural Networks

Tae-Hoon Lee, Min-Soo Kim[*]

Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
{th.lee, minsoo.k}@kaist.ac.kr

Abstract

Mixed-Integer Linear Programming (MILP) is an optimization technique widely used in various fields. Existing *end-to-end learning* methods for MILP generate values for a subset of decision variables and delegate the remaining problem to traditional MILP solvers. However, this approach does not guarantee solution feasibility (i.e., satisfying all constraints) due to inaccurate predictions and primarily focuses on prediction for binary decision variables. When addressing MILP involving non-binary integer variables using machine learning (ML), feasibility issues can become even more pronounced. Since finding an optimal solution requires satisfying all constraints, addressing feasibility is critical. To overcome these limitations, we propose a novel reinforcement learning (RL)-based solver that interacts with MILP to incrementally discover better feasible solutions without relying on traditional solvers. We design reward functions tailored for MILP, which enable the RL agent to learn relationships between decision variables and constraints. Furthermore, we leverage a Transformer encoder-based graph neural network (GNN) to effectively model complex relationships among decision variables. Our experimental results demonstrate that the proposed method can solve MILP problems and find near-optimal solutions without delegating the remainder to traditional solvers. The proposed method provides a meaningful step forward as an initial study in solving MILP problems entirely with ML in an end-to-end manner.

1 Introduction

The traveling salesman problem (TSP) and the knapsack problem are representative examples of combinatorial optimization (CO) problems that have been extensively studied in operations research and computer science. CO addresses mathematical optimization problems that aim to minimize or maximize the value of a specific objective function. If the objective function and constraints of CO are linear, it is called linear programming (LP). If some decision variables in LP must take integer values, it becomes Mixed-Integer Linear Programming (MILP) (Bengio, Lodi, and Prouvost 2021). MILP is widely used to model various problems (Zhang et al. 2023) such as logistics (Kweon et al. 2024), path planning (Zuo et al. 2020), and energy systems (Ren and Gao 2010).

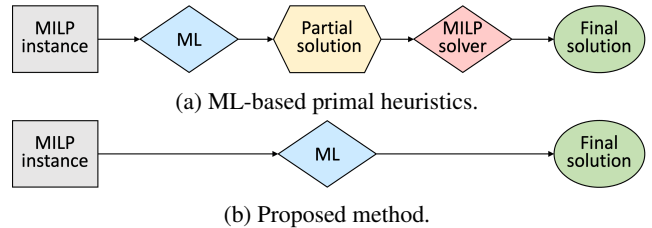(a) ML-based primal heuristics.

(b) Proposed method.

Figure 1: Solving MILP with *end-to-end learning* methods.

ML methods that directly learn and generate solutions for CO problems are categorized as an *end-to-end learning* approach (Bengio, Lodi, and Prouvost 2021). ML-based primal heuristics are considered as *end-to-end learning* (Han et al. 2023), which aim to quickly find initial solutions to MILP by reducing the search space (Achterberg, Berthold, and Hendel 2012). Unlike traditional primal heuristics, which rely on expert knowledge and hand-crafted designs (Bengio, Lodi, and Prouvost 2021), the ML-based one leverages similar patterns shared by MILP instances generated from a specific distribution (Gasse et al. 2022). Figure 1(a) illustrates how ML-based primal heuristics solve MILP. A trained ML model generates a partial solution for a subset of integer variables in a given MILP instance. The partial solution is passed to a traditional optimizer (e.g., Gurobi) that optimizes the remaining problem to obtain the final solution.

Existing ML-based primal heuristics have demonstrated their ability to quickly find good solutions by reducing the problem space (Gasse et al. 2019; Nair et al. 2020; Yoon 2022; Han et al. 2023; Cantürk et al. 2024). However, inaccurate ML predictions hinder solution feasibility (i.e., satisfying all constraints), which emphasizes the need to consider feasibility (Han et al. 2023). The infeasibility prevents reaching the optimal solution, posing a significant obstacle to solving MILP. Moreover, most ML-based primal heuristics (Gasse et al. 2019; Yoon 2022; Han et al. 2023; Cantürk et al. 2024) focus solely on predicting values for binary decision variables. However, many real-world problems involve non-binary integer (integer, for short) variables such as logistics (Kweon et al. 2024), maritime transportation (Papageorgiou et al. 2014), and energy systems (Ren and Gao 2010). This highlights the need for techniques capable of effectively handling integer variables.

---

[*]Corresponding author.

Infeasibility caused by incorrect ML predictions can be more pronounced for integer variables due to their broader prediction range than binary variables. For instance, imagine a naive ML approach that uses regression to predict the value of an integer variable x_i within the range $[0, 1000]$. The maximum possible prediction error for a binary variable, with a range of $[0, 1]$, is 1. However, since the range of x_i is significantly broader, the maximum prediction error for x_i would also be broader. Moreover, additional errors can occur due to rounding that makes the predicted values integers. Therefore, a more accurate ML-based method for integer variables is required. Nair et al. have proposed a method to handle integer variables by representing the values in binary format. To represent x_i as a binary sequence, the required sequence length would be $\lceil \log_2(1000) \rceil = 10$. Thus, predicting a single decision variable requires multiple dimensions, and the upper/lower bounds of a variable may even be infinite in some cases. It calls for more efficient methods to handle integer decision variables.

To address these limitations, we propose an RL-based method for solving MILP. Figure 1(b) illustrates how the proposed solver derives the final solution for an MILP instance. Unlike the existing *end-to-end learning* methods that focus on predictions for a subset of variables in MILP, the proposed method generates complete feasible solutions. Rather than directly predicting the exact values of integer variables, we adopt an indirect approach where the ML model determines whether to increase, decrease, or retain the current value of each decision variable. To guide these decisions, we design an RL system tailored to MILP, enabling the agent to effectively learn the relationships between decision variables and constraints. The solution search process is divided into two phases: before and after finding the first feasible solution of MILP. We design reward functions for each phase to align with their objectives. During training, the RL agent is updated based on the degree of constraint violations and improvements in the objective value. To capture the relationships among all decision variables effectively, we adopt a Transformer encoder-based GNN as the agent's architecture. The trained agent processes unseen MILP instances and incrementally improves the solution using the proposed local search strategy. Experimental results demonstrate that the proposed model achieves optimal solutions on a small dataset and finds near-optimal solutions with roughly 1% of the optimal for a larger dataset. Our main contributions are summarized as follows:

- We propose a novel RL-based MILP solver (RL-MILP solver) capable of finding complete feasible solutions, rather than a partial solution.

- We design an RL system that enables the agent to effectively learn the relationships between decision variables and constraints by interacting with MILP problems.

- We propose a Transformer encoder-based GNN as the agent's architecture, designed to effectively capture the complex relationships among decision variables in MILP.

2 Preliminaries

2.1 Mixed-Integer Linear Programming

Mixed-Integer Linear Programming (MILP) is a mathematical optimization problem that minimizes (or maximizes) a linear objective function while satisfying linear constraints and the integrality requirements for some decision variables (Bertsimas and Tsitsiklis 1997). The standard form of MILP is as follows:

$$\text{minimize } \mathbf{c}^T \mathbf{x} \tag{1}$$
$$\text{subject to } \mathbf{A}\mathbf{x} \le \mathbf{b} \tag{2}$$
$$x_i \in \mathbb{Z}, \forall i \in I \tag{3}$$
$$l_i \le x_i \le u_i, \forall i \tag{4}$$

where $\mathbf{x} \in \mathbb{R}^n$ is a column vector of n decision variables, $\mathbf{c} \in \mathbb{R}^n$ is a column vector of coefficients for the objective function, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the constraint coefficient matrix, $\mathbf{b} \in \mathbb{R}^m$ is a column vector of the right-hand side of the constraints, I is the index set of integer decision variables, l_i/u_i denotes the lower/upper bounds for each decision variable x_i. The goal of MILP is to find the optimal solution, and for a minimization problem, this corresponds to a feasible solution $\mathbf{x}$ that minimizes $obj = \mathbf{c}^T \mathbf{x}$. A feasible solution is defined as a solution $\mathbf{x}$ that satisfies all constraints (Eqs. 2-4). Integrality requirements (Eq. 3) make the solution space of MILP discrete. As the number of integer variables increases, the possible combinations grow exponentially, increasing the difficulty of solving MILP within polynomial time. LP-relaxation refers to the technique of removing the integrality requirement in MILP, which transforms it into an LP problem solvable in polynomial time (Bertsimas and Tsitsiklis 1997). LP-relaxation is widely used in traditional algorithms, such as Branch-and-Bound (Land and Doig 2010) and Feasible Pump (Fischetti, Glover, and Lodi 2005), to obtain an initial solution for MILP.

All MILP problems can be transformed into the form shown in Equations 1-4 (Bertsimas and Tsitsiklis 1997). Let $\mathbf{a}_i^T$ denote a row vector of a single constraint, $\mathbf{A} = (\mathbf{a}_1^T, \ldots, \mathbf{a}_m^T)$, and $\mathbf{b} = (b_1, \ldots, b_m)$. An equality constraint $\mathbf{a}_i^T \mathbf{x} = b_i$ is equivalent to two inequality constraints ($\mathbf{a}_i^T \mathbf{x} \ge b_i$ and $\mathbf{a}_i^T \mathbf{x} \le b_i$). Moreover, $\mathbf{a}_i^T \mathbf{x} \ge b_i$ is equivalent to $-\mathbf{a}_i^T \mathbf{x} \le -b_i$. Similarly, maximizing $\mathbf{c}^T \mathbf{x}$ is equivalent to minimizing $-\mathbf{c}^T \mathbf{x}$. For example, a problem with the objective function maximizing $+2x_1 + 3x_2 - x_3$ can be reformulated as minimizing $-2x_1 - 3x_2 + x_3$, and the constraint $-x_1 + 2x_3 \ge -5$ can be rewritten as $x_1 - 2x_3 \le 5$. Thus, regardless of the set of constraints or the optimization direction, all cases can be transformed into the standard form given in Equations 1-4. Consequently, we focus only on solving MILP problems that follow this standard form.

2.2 Graph Representation of MILP

Studies on ML-based primal heuristics, which assist traditional optimizers (e.g., Gurobi, SCIP), represent MILP instances as bipartite graphs (Gasse et al. 2019; Nair et al. 2020; Yoon 2022; Han et al. 2023; Cantürk et al. 2024). In a bipartite graph representation of MILP, one set of nodes is for constraints, and the other is for decision variables. An

edge connects a decision variable node to a constraint node only if the variable appears in the corresponding constraint. For example, in Figure 2(a), the decision variable x_3 appears in the constraint $\mathbf{a_2}$. Therefore, in the MILP bipartite graph, the variable node for x_3 is connected to the constraint node for $\mathbf{a_2}$.

2.3 Graph Neural Networks

Message Passing Neural Network   The Message Passing Neural Network (MPNN) (Gilmer et al. 2017) is a general framework for message-passing-based GNNs. Widely used architectures such as GCN (Kipf and Welling 2017), GAT (Veličković et al. 2018), and GIN (Xu et al. 2019) are GNN architectures based on MPNN. Given a graph G, the new representation $h_v^{(k+1)}$ for a target node v is obtained as follows:

$$m_{vu}^{(k+1)} = msg(h_v^{(k)}, h_u^{(k)}, e_{vu}), \forall e_{vu} \in \mathcal{E}$$
$$g_v^{(k+1)} = agg(\{m_{vu}^{(k+1)} | u \in \mathcal{N}(v)\}), \forall v \in \mathcal{V} \quad (5)$$
$$h_v^{(k+1)} = update(h_v^{(k)}, g_v^{(k+1)}), \forall v \in \mathcal{V}$$

where $h_v^{(0)}$ is the initial feature vector of node v, and $msg(\cdot)$, $agg(\cdot)$, and $update(\cdot)$ are the message-passing, aggregation, and update functions, respectively. $msg(\cdot)$ generates a message $m_{vu}^{(k+1)}$ using the representation of target node $h_v^{(k)}$, the neighbor node $h_u^{(k)}$, and the edge feature e_{vu}. $agg(\cdot)$ aggregates the messages $m_{vu}^{(k+1)}$ generated by $msg(\cdot)$ for each target node. $update(\cdot)$ updates the target node v to a new representation $h_v^{(k+1)}$ by combining the aggregated information $g_v^{(k+1)}$ with the previous embedding $h_v^{(k)}$. The embedding $h_v^{(k+1)}$ is either used for message passing in the next layer or for prediction tasks.

Since edges exist only between nodes from different sets (i.e., variable-constraint) in an MILP bipartite graph, obtaining a new representation for a decision variable requires two rounds of message passing. As shown in Figure 2(b), a new representation for the decision variables is obtained by performing one constraint-side convolution and one variable-side convolution after applying initial embeddings.

Transformer for Graphs   Graph Transformers are models that extend the Transformer architecture (Vaswani 2017) to handle graph data. Originally designed as a sequence-to-sequence model for machine translation, the Transformer has achieved significant success in various domains such as NLP and computer vision. Recently, there have been numerous efforts to adapt Transformers for the graph domain (Lin et al. 2022; Min et al. 2022). The attention mechanism of Transformers enables each node to attend to every node, which allows the model to effectively learn relationships between distant nodes (Wu et al. 2021).

In contrast, MPNN-based GNNs receive messages from the neighbor node, which is suitable for learning local structural information. However, they struggle to capture relationships between distant nodes (Zhang et al. 2020; Wu et al. 2021). To propagate messages from nodes that are K hops away, an MPNN-based GNN requires K layers. However,
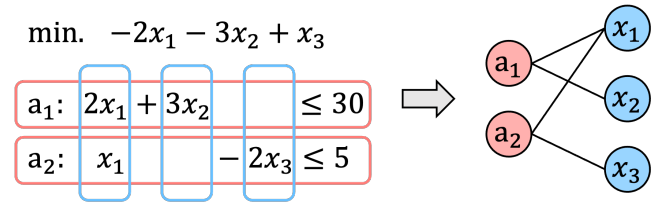
(a) Example of a bipartite graph representation for an MILP instance with three variables and two constraints.
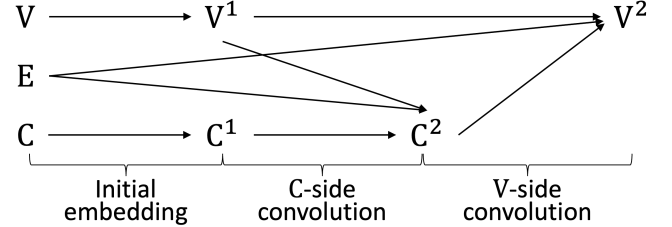
(b) GCN for bipartite graph representation of MILP. V: Variable nodes, C: Constraint nodes, E: Edge features and adjacency matrix.

Figure 2: Illustration of MILP bipartite graph representation and its GCN architecture.

deeper layers can lead to the oversmoothing, where nodes have similar and indistinguishable representations (Li, Han, and Wu 2018; Wu et al. 2021; Min et al. 2022).

A GNN for MILP may require deep layers to capture the relationships between variables that influence each other across multiple constraints. For instance, as illustrated in Figure 2(a), the variables x_2 and x_3 are 4 hops apart (x_2 - $\mathbf{a_1}$ - x_1 - $\mathbf{a_2}$ - x_3). Although x_2 and x_3 do not appear in the same constraint, they are connected via x_1, which appears in both $\mathbf{a_1}$ and $\mathbf{a_2}$. A change in the value of x_2 can affect x_1, which may subsequently affect x_3. Thus, capturing the relationship between these decision variables is essential. Propagating messages from x_2 to x_3 requires four graph convolution layers. However, even shallow networks with 2-4 GNN layers can suffer from the oversmoothing (Wu et al. 2023). To address this, we utilize a Transformer encoder-based GNN, which can effectively learn relationships between all nodes, regardless of their distance.

3   Methodology

This section provides a detailed introduction to our RL-MILP solver. Figure 3 demonstrates how the proposed solver incrementally improves feasible solutions. The first step involves transforming the MILP instance in its standard form (Eqs. 1-4) into a bipartite graph. The MILP bipartite graph serves as input to the GNN-based agent. In the second step, the agent trained with an RL algorithm selects actions estimated to yield high rewards. The agent decides for each variable whether to increase, decrease, or retain its value. In the third step, the selected actions are applied to derive a solution for the given instance. The final solution is updated if the new feasible solution $\mathbf{x}$, obtained from the selected actions, is better than the best feasible solution found so far.
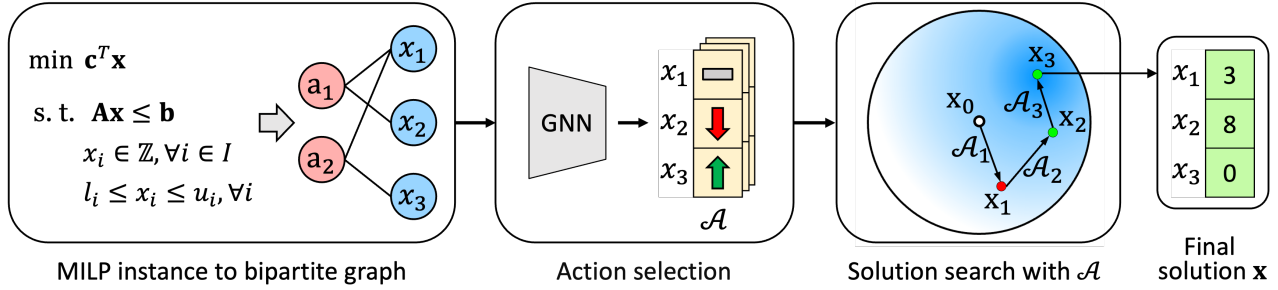
Figure 3: The overview of our approach. Green ↑, red ↓, and gray − represent an increase, a decrease, and no change in decision variable values, respectively. The white point, red point, and green points denote the initial solution, an infeasible solution, and feasible solutions, respectively. The darkest blue area marks the optimal solution.
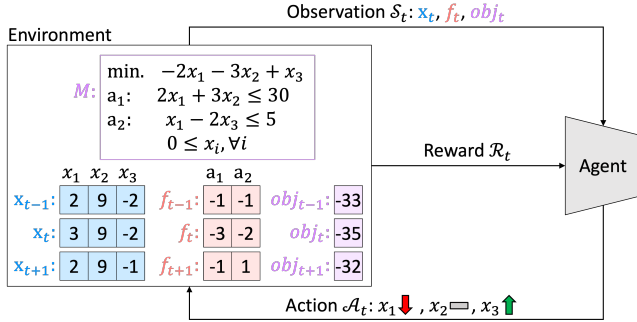

Figure 4: Diagram of reinforcement learning for MILP.

3.1 Reinforcement Learning for MILP

In our scenario, RL aims to train the agent to make decisions that maximize rewards while interacting with a given MILP instance. Figure 4 illustrates how the RL agent interacts with an example MILP instance, where $\mathcal{S}_t$, $\mathcal{A}_t$, and $\mathcal{R}_t$ denote the observation, the set of actions, and reward at timestep t, respectively. The given instance M acts as the environment with which the RL agent interacts. Based on the set of actions $\mathcal{A}_t = (a_{t,1}, \ldots, a_{t,n})$ selected by the agent for n variables, the solution $\mathbf{x}_{t+1}$ is updated. This update, in turn, changes the left-hand side of the constraints lhs_{t+1}, the feasible state f_{t+1}, and the objective value obj_{t+1}. We address details regarding f_{t+1} in the next section (Observation in Section 3.1). In the next timestep, the agent faces the new environment changed by its previous actions. The agent selects a new action set $\mathcal{A}_{t+1}$ that is expected to maximize rewards in the new environment. By comparing the actual reward $\mathcal{R}_{t+1}$ obtained from the action with the expected reward, the agent updates its policy π.

Observation In RL for solving MILP, we define the solution $\mathbf{x}_t$, the feasible state f_t, and the objective value obj_t at timestep t as the observation $\mathcal{S}_t = (\mathbf{x}_t, f_t, obj_t)$. The solution $\mathbf{x}_t$ is derived by updating the values of each variable based on the agent's actions $\mathcal{A}_{t-1}$. For example shown in Figure 4, assuming $\mathcal{A}_{t-1} = (a_{t-1,1} = +1, a_{t-1,2} = +0, a_{t-1,3} = +0)$, then $\mathbf{x}_{t-1}$ is updated to $\mathbf{x}_t = (x_{t,1} = 3, x_{t,2} = 9, x_{t,3} = -2)$. With the updated $\mathbf{x}_t$, the new

$lhs_t = \mathbf{A}\mathbf{x}_t$ and $obj_t = \mathbf{c}^T\mathbf{x}_t$ are calculated.

f_t is obtained from the difference between $\mathbf{b}$ and lhs_t (i.e., $f_t = \mathbf{b} - lhs_t$). Each element of the f_t vector indicates whether $\mathbf{x}_t$ satisfies the corresponding constraint. A positive element in f_t indicates that the corresponding constraint is satisfied, whereas a negative element means that the constraint is violated. To ensure stable training, f_t is scaled by the sum of $|\mathbf{b}|$ and $|lhs_t|$ (i.e., $f_t = (\mathbf{b} - lhs_t)/(|\mathbf{b}| + |lhs_t|)$). However, for simplicity, this paper explains f_t without scaling. For example, in Figure 4, the lhs_{t+1} for constraints $\mathbf{a}_1$ and $\mathbf{a}_2$ derived from $\mathbf{x}_{t+1}$ are 31 and 4, respectively. Thus, we have $f_{t+1} = (\mathbf{a}_1 = 30 - 31 = -1, \mathbf{a}_2 = 5 - 4 = 1)$. From the elements of f_{t+1}, we can infer that $\mathbf{x}_{t+1}$ violates $\mathbf{a}_1$ but satisfies $\mathbf{a}_2$.

Action The RL agent for solving MILP selects the optimal action for each decision variable to maximize the reward in a given situation. At timestep t, the agent determines the change in each variable's value based on the observation $\mathcal{S}_t = (\mathbf{x}_t, f_t, obj_t)$. For each variable, the agent can take one of three types of actions: increase (↑), decrease (↓), or retain (−), as shown in Figure 4. In this study, the magnitude of change for both increases and decreases is set to 1, resulting in three possible actions for each variable.

Reward The reward system for the RL agent solving MILP is designed in two phases: $phase1$ and $phase2$. $phase1$ covers the period until the first feasible solution is found, while $phase2$ starts after the first feasible solution is found. In $phase1$, the solver aims to find a solution that satisfies all the constraints of the given MILP instance. The reward in $phase1$ is calculated based on the degree of improvement or deterioration in each constraint and whether
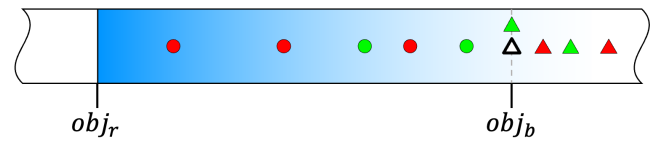

Figure 5: Illustration of reward function in $phase2$. ○: inside, △: outside, green: feasible, red: infeasible, white △ : obj_b.

the bounds of the decision variables are satisfied, as follows:

$$\mathcal{R}_t = \sum_{j=1}^{m} (f_{t+1,j} - f_{t,j}) - \sum_{i=1}^{n} \mathbb{I}\,(x_{t,i} \notin [l_i, u_i]) \qquad (6)$$

where f_t denotes the m-dimensional feasible state for the m constraints, and $x_{t,i}$ is the value of decision variable i at timestep t. The reward depends on the degree of improvement or deterioration in each constraint caused by the agent's actions $\mathcal{A}_t$ (first term in Eq. 6). Using f_t and f_{t+1} given in Figure 4, the calculated reward is $\{-1 - (-3)\} + \{1 - (-2)\} = 5$. Additionally, the agent is penalized by the number of decision variables that violate their bound conditions (second term in Eq. 6). As shown in Figure 4, since the lower bound for all decision variables is 0, the bound penalty at timestep t is -1 due to $x_{t,3} = -2$. Hence, the reward $\mathcal{R}_t$ is obtained by summing the two terms: $5 + (-1) = 4$.

In $phase2$, the solver aims to find feasible solutions that yield a better objective value than the best one previously found. The reward is calculated as follows:

$$\mathcal{R}_t = \begin{cases} 1 - \delta_b, & \text{if } obj_t \in (obj_r, obj_b), \\ -\delta_b, & \text{if } obj_t \notin (obj_r, obj_b) \wedge \text{infeasible}, \\ 0, & \text{if } obj_t \notin (obj_r, obj_b) \wedge \text{feasible}. \end{cases} \quad (7)$$

$$\mathcal{R}_t += C_{\text{reward}}, \quad \text{if feasible} \wedge obj_t < obj_b \qquad (8)$$

$$\delta_b = \frac{|obj_t - obj_b|}{|obj_b|} \qquad (9)$$

where obj_r serves as the lower bound that is the objective value of the LP-relaxation of the original MILP problem, and obj_b denotes the smallest objective value found so far, C_{reward} denotes a positive constant which is set to $+1$ in our study. The value of δ_b, ranging from 0 to 1, indicates how close obj_t is to obj_b (Eq. 9). Smaller δ_b values indicate greater proximity between the two.

To enhance understanding, we explain the reward function in $phase2$ (Eq. 7) with Figure 5. If a better (i.e., lower) objective value than obj_b exists, it must lie between the lower bound obj_r and obj_b. Thus, the agent should explore solutions in this interval to obtain a better obj_t. Getting closer to obj_r improves the obj value, but finding feasible solutions becomes harder in this region because of the tight constraints. Therefore, the agent is encouraged to search for solutions closer to obj_b, where finding feasible solutions is relatively easier. For this reason, within the interval, higher rewards are given as obj_t gets closer to obj_b (Case 1 in Eq. 7). Regardless of whether $\mathbf{x}_t$ is feasible, a reward is given if obj_t lies within the interval. In Figure 5, the red and green circles correspond to this first case. We regard the agent's actions that result in obj_t falling outside the interval as incorrect. For infeasible solutions $\mathbf{x}_t$ outside the interval, a penalty is given proportional to the distance of obj_t from obj_b (Case 2 in Eq. 7). In Figure 5, the red triangle represents this second case. On the other hand, penalizing feasible solutions outside the interval could undermine the agent's learning from $phase1$, where the goal is to find feasible solutions. To preserve consistency in encouraging feasible solution discovery across $phase1$ and $phase2$, no penalties are given for feasible solutions outside the interval (Case 3 in Eq. 7). In Figure 5, the green triangle corresponds to this third case. If the agent finds a better feasible solution within the interval, an additional reward of C_{reward} is given (Eq. 8), and it corresponds to the green circle in Figure 5. Finally, in both $phase1$ and $phase2$, if the agent does not perform any exploration (i.e., no change ($-$) for all variables), a penalty of -10 is added to $\mathcal{R}_t$ to encourage exploration.

Learning Algorithm For training the RL agent, we utilize the Actor-Critic algorithm (Mnih et al. 2016) which has been effective in solving CO problems (Bello et al. 2016; Hubbs et al. 2020). The Actor-Critic algorithm combines a policy-based actor and a value-based critic. The policy-based actor aims to optimize the agent's parameters θ to maximize the expected cumulative reward. The actor directly learns a policy $\pi(\mathcal{A} \mid \mathcal{S}; \theta)$, which maps the observation $\mathcal{S}$ into a probability distribution over actions $\mathcal{A}$. Meanwhile, the value-based critic evaluates the value of the action given an observation and learns to select actions that maximize cumulative rewards. The critic learns a value function $Q(\mathcal{S}, \mathcal{A})$ estimating the expected cumulative reward for a given $\mathcal{S}$-$\mathcal{A}$ pair.

At each timestep t, the agent observes the environment's state $\mathcal{S}_t$ and decides actions to take using the policy π. When the agent executes a set of actions $\mathcal{A}_t$, the environment reacts by providing a new observation $\mathcal{S}_{t+1}$ and a reward $\mathcal{R}_t$ associated with the action. The actual (observed) reward $\mathcal{R}_t$ is compared with the estimated reward $Q(\mathcal{S}_t, \mathcal{A}_t)$. The actor is trained to encourage actions that yield actual rewards higher than the estimated reward while suppressing actions that result in lower actual rewards. Meanwhile, the critic is trained to minimize the difference between the actual reward $\mathcal{R}_t$ and the estimated reward $Q(\mathcal{S}_t, \mathcal{A}_t)$ to provide more accurate feedback to the actor.

To enable the RL agent to learn optimal actions across various environments, we expose the agent to a diverse set of MILP instances. Details on the random generation of MILP instances are provided in Section 4.1 The agent uses instances as the environment and continues training until a termination condition is met. In this study, the agent terminates training for an instance if it either reaches a predefined step limit or fails to find a better solution in $phase2$ over a predefined number of steps. Once the agent meets the termination condition, it switches to a new MILP instance.

3.2 GNN Architecture of RL Agent

For the agent's architecture, we adopt a hybrid approach that combines an MPNN-based GNN with a Transformer encoder (Rong et al. 2020; Wu et al. 2021). Figure 6 illustrates the architecture of the RL agent for solving MILP problems. In our study, the MPNN-based GNN captures local information specific to each decision variable. Meanwhile, the Transformer encoder utilizes the captured local information, along with additional MILP features, as inputs to learn global information across all variables. We encode the following features as inputs to the GNN: the objective value obj_t, the coefficients of the objective function $\mathbf{c}$, the constraint coefficients $\mathbf{A}$, the right-hand side of the constraints $\mathbf{b}$, the solution values $\mathbf{x}_t$, and the bound features bnd_lim for the variables' lower and upper bounds.
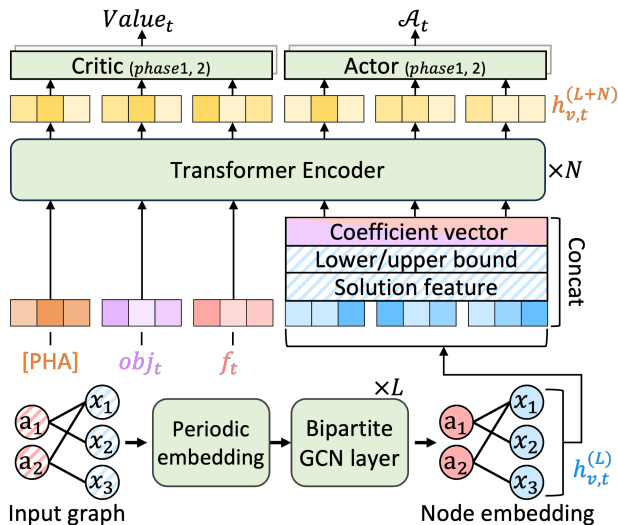
Figure 6: The architecture of the GNN for the Actor-Critic agent solving MILP.

The MPNN-based GNN aims to capture local information for each decision variable. In an MILP graph, local information can include whether the constraints involving a given decision variable are satisfied. To learn feasibility relationship between variables and constraints, we use $\mathbf{x}_t$, $\mathbf{b}$, and $\mathbf{A}$ as the features for variable nodes, constraint nodes, and edges, respectively. Input feature scaling is necessary to ensure stable training. We apply equilibration scaling (Tomlin 1975) to scale $\mathbf{A}$. Equilibration scaling relies on the principle that multiplying each constraint by a non-negative scalar does not change the optimization results of the original MILP instance. We scale $\mathbf{A}$ to the range $[-1, 1]$ by dividing each constraint by its own largest absolute coefficient. For $\mathbf{x}_t$ and $\mathbf{b}$, whose bounds are difficult to define and thus challenging to scale, we employ a periodic activation-based module, called Periodic Embedding (PE), to embed scalar values into vectors. PE has been shown to improve the performance of DL models in tasks involving numerical feature processing (Gorishniy, Rubachev, and Babenko 2022), such as house price prediction (Pace and Barry 1997) and income prediction (Kohavi et al. 1996). PE is formulated as follows:

$$\text{Periodic}(x) = \text{concat}[\sin(v), \cos(v)],$$
$$v = [2\pi c_1 x, \ldots, 2\pi c_k x].$$
(10)

where x is a scalar value, and c_i is trainable parameters. The constraint-wise scaled $\mathbf{A}$ serves as the weighted edge between variable and constraint nodes in the MILP graph. The graph passes through L layers of bipartite GCN, after which the resulting node embeddings $h_{v,t}^{(L)}$ are fed into the Transformer encoder as inputs.

We utilize the Transformer encoder to enable the RL agent to capture global information about MILP problems. Global information in an MILP graph may include relationships among decision variables connected through multiple constraints, as well as the feasibility and optimality of the solution. As shown in Figure 6, the transformer encoder takes a

phase token [PHA], PE-encoded (Eq. 10) obj_t, f_t, and $\tilde{h}_{v,t}^{(L)}$ as input. The vector $\tilde{h}_{v,t}^{(L)} = [h_{v,t}^{(L)}, h_{v,t}^{(0)}, (\mathbf{c}|\mathbf{A}^T), bnd\_lim]$ represents the embeddings for decision variables, where $[\ldots]$ denotes the concatenation operation and $(\mathbf{c}|\mathbf{A}^T)$ is the coefficient matrix containing structural information of the graph. bnd_lim is a binary feature set to 1 if the variable lies on the boundary of its lower or upper bound, and 0 otherwise. For example, if the values of two variables x_j and x_k are 0 and 5 with a lower bound of 0, respectively. In this case, the bnd_lim for x_j and x_k would be 1 and 0, respectively.

The [PHA] token indicates the phase in which the agent is currently operating. As mentioned in Section 3.1, the goals and reward systems differ between the two phases, so the reward for an agent's action can vary depending on the phase. If the agent is unaware of the current phase, it may take wrong actions that are unsuitable for the situation. For example, in $phase2$, the agent might take wrong actions by staying outside the interval rather than striving to find better feasible solutions within the interval, even though it has already found a feasible solution in $phase1$ (shown as the green △ in Figure 5). To prevent such behavior, we introduce the [PHA] token to inform the agent of the current phase.

The Transformer encoder leverages the attention mechanism to learn relationships among decision variables and between variables and constraints. The encoder's output, $h_{v,t}^{(L+N)}$, serves as the input to the Actor layer, which generates $\mathcal{A}_t$. The Critic layer uses the final embeddings for [PHA], obj_t, and f_t as inputs to estimate the reward resulting from the action $\mathcal{A}_t$. The reason for taking f_t and obj_t in the Critic layer is that rewards are given based on the improvements/deteriorations of feasibility and the discovery of better obj values. Since the reward system is divided into $phase1$ and $phase2$ depending on whether the first feasible solution has been found, the Critic and Actor layers are also separated for each phase. All layers, except for the Critic and Actor layer, share parameters across the two phases.

3.3 Solution Search Strategy

Our solution search strategy of the agent is inspired by the concept of local search, a widely used heuristic algorithm. Local search aims to explore the neighborhood of the current solution to find a good solution for the MILP problem (Hillier and Lieberman 2015). While local search does not guarantee solving the given problem in polynomial time, empirical evidence suggests that it is effective at quickly finding good solutions (Bertsimas and Tsitsiklis 1997). Algorithm 1 provides the pseudo-code for the proposed search algorithm. This algorithm leverages a GNN-based agent parameterized by θ to guide the search process and aims to find the optimal (or near-optimal) solution.

This algorithm is designed to iteratively search for the optimal solution of a given instance M by exploring the neighborhood of the best solution found so far during the inference phase. The solution search process proceeds as follows: From the LP-relaxation of instance M, the initial solution $\mathbf{x}_0$ and the objective value obj_r, which serves as the lower bound, are obtained (Line 1). Since the values of decision variables obtained from LP-relaxation are continuous, $\mathbf{x}_0$ is

Algorithm 1: Solution Search Algorithm

Input: Problem instance M, Local region size Δ
Parameter: Agent parameters θ
Output: Best feasible solution $\mathbf{x}_b$

1: Obtain an initial solution $\mathbf{x}_0$ and a lower bound obj_r
2: $t \leftarrow 0$
3: $obj_b \leftarrow \infty$
4: $\mathbf{x}_b \leftarrow \mathbf{x}_0$
5: **while** time_limit not reached **do**
6:    Observe the current observation $\mathcal{S}_t$ from M
7:    Select a set of actions $\mathcal{A}_t$ using $\pi(\mathcal{A}_t \mid \mathcal{S}_t; \theta)$
8:    Obtain the neighbor $\mathbf{x}_{t+1}$ by applying $\mathcal{A}_t$ to $\mathbf{x}_t$
9:    $obj_{t+1} \leftarrow \mathbf{c}^T \mathbf{x}_{t+1}$
10:   **if** $\mathbf{x}_{t+1}$ is feasible **and** $obj_{t+1} < obj_b$ **then**
11:      $obj_b \leftarrow obj_{t+1}$
12:      $\mathbf{x}_b \leftarrow \mathbf{x}_{t+1}$
13:   **else if** any element of $|\mathbf{x}_b - \mathbf{x}_{t+1}| > \Delta$
        or $obj_{t+1} \notin (obj_r, obj_b)$ **then**
14:      $\mathbf{x}_{t+1} \leftarrow \mathbf{x}_b$
15:   **end if**
16:   $t \leftarrow t + 1$
17: **end while**
18: **return** $\mathbf{x}_b$

rounded to remove decimal points, which often results in an infeasible solution. The iteration counter t is initialized to 0, the best objective value obj_b is initialized to infinity (∞), and the best solution $\mathbf{x}_b$ is set to the initial solution $\mathbf{x}_0$ (Lines 2-4). The algorithm repeats the following solution search steps until a predefined time limit is reached (Lines 5-17): The agent selects actions $\mathcal{A}_t$ that are likely to yield higher rewards in the current situation (Lines 6-7). Based on $\mathcal{A}_t$, the next solution $\mathbf{x}_{t+1}$ and its objective value obj_{t+1} are obtained (Lines 8-9). If $\mathbf{x}_{t+1}$ is feasible and obj_{t+1} is better (i.e., lower) than obj_b, then obj_b and $\mathbf{x}_b$ are updated to the new obj_{t+1} and $\mathbf{x}_{t+1}$, respectively (Lines 10-12). Updating $\mathbf{x}_b$ implies that the center of the local region is also updated. If $\mathbf{x}_{t+1}$ falls outside the local region or the target search interval, it is moved back to the center of the local region (i.e., $\mathbf{x}_b$) (Lines 13-15). After completing a search step, t is incremented by 1 (Line 16). When the predefined time limit is reached, the algorithm terminates and returns $\mathbf{x}_b$ (Line 18).

Parameter	Distribution
c	randint$[-10, 1]$
A	randint$[1, 10]$ with density $\rho = 0.1$
b	$\mathbf{A}\xi + \epsilon$, where
	$\xi_i \sim$ randint$[1, 10], \forall i = 1, \ldots, n$ and
	$\epsilon_j \sim$ randint$[1, 10], \forall j = 1, \ldots, m$
I	$\{x \mid x \in \mathbb{N}, 1 \leq x \leq n\}$
l_i	$0, \forall i = 1, \ldots, n$
u_i	$\infty, \forall i = 1, \ldots, n$

Table 1: Parameters for MILP instance generation.

4 Experiments

4.1 Experimental Setup

Dataset Collection We generated MILP instances by referencing the method for random instance generation described in (Qi, Wang, and Shen 2021). Table 1 summarizes the parameters used for instance generation. Considering that the ratio of non-zero coefficients ρ in typical LP problems is less than 5% (Hillier and Lieberman 2015), we set a higher density of 10% to promote more interactions between variables in constraints. In this study, we assume a special case of MILP problems where all decision variables are non-binary integers (integer, for short) to focus on handling integer variables. According to the default settings of the SOTA optimizer Gurobi, the lower bound l_i and upper bound u_i for decision variables are set to 0 and ∞, respectively. Datasets are named in the format D_{n*m}, where n and m are the number of decision variables and constraints, respectively. We use two datasets for our experiments: D_{9*18}, which has the same size as the largest dataset used in the prior research (Qi, Wang, and Shen 2021), and D_{100*50}, which is approximately 30 times larger than D_{9*18}. Training data is generated on-the-fly whenever the agent completes training on an instance. In contrast, for the testing phase, we use 100 pre-generated instances for each dataset.

Compared Methods We evaluated the performance of the proposed method by comparing it with Gurobi known as a SOTA commercial optimizer. In addition, to verify the effectiveness of the Transformer encoder, we compared several variations of the RL agent's architecture. The first variation is a Multi-Layer Perceptron (MLP), a basic architecture composed of multiple fully connected layers. MLP has a simple structure and is suitable for typical data processing. The second variation is a Convolutional Neural Network (CNN), an architecture designed to effectively extract local features of the data. CNN is well-suited for structured data like images, and in our experiments, each channel in the CNN corresponds to a decision variable. The MLP and CNN architectures replace the Transformer encoder in Figure 6 and are trained alongside the MPNN-based bipartite GCN, which we call M_MLP and M_CNN, respectively. M_MLP and M_CNN use the same input data as the proposed method and have independent Critic/Actor layers for $phase1$ and $phase2$ likewise. We extended the Actor-Critic algorithm implemented in PyTorch (Kostrikov 2018) to suit MILP problems. All ML variations were trained using the same algorithm, with 500 and 5,000 parameter updates for the D_{9*18} and D_{100*50} datasets, respectively. For comparison, each method solves minimization problems for each instance within 100 seconds.

Metrics To evaluate the compared methods from the perspective of feasibility, we introduce two metrics: *Feasible Rate* (FR) and *First Feasible Solution Time* (FFST). FR represents the ratio of test instances where a method successfully found a feasible solution. FFST measures the amount of time (in seconds) that the method takes to obtain the first feasible solution. Additionally, to evaluate the solution quality of the methods, we use the metric *Relative Primal*

Gap (RP_Gap). RP_Gap quantifies the difference between the best objective value found by the method and the best objective value obtained across all methods (Gasse et al. 2022). RP_Gap is calculated as follows:

$$\text{RP\_Gap} = \frac{|obj_b - obj^*|}{|obj^*|} * 100 \qquad (11)$$

where obj_b is the best objective value found by the evaluated method, and obj^* is the best across all compared methods. RP_Gap indicates how close obj_b is to obj^*.

Dataset	Method	Metric		
		FR↑	FFST↓	RP_Gap↓
D_{9*18}	Gurobi	100/100	< 1	0
	M_MLP	100/100	0.05	4.45
	M_CNN	100/100	0.05	1.90
	Ours	100/100	0.07	0
D_{100*50}	Gurobi	100/100	< 1	0.01
	M_MLP	100/100	0.06	1.60
	M_CNN	100/100	0.07	1.96
	Ours	100/100	0.06	1.33

Table 2: Performance comparison with compared methods in terms of *Feasible Rate* (FR), *First Feasible Solution Time* (FFST), and *Relative Primal Gap* (RP_Gap).

4.2 Main Results

Table 2 shows the performance of the compared methods. The proposed model and its variants (M_MLP and M_CNN) successfully found feasible solutions for all instances in both datasets. It indicates that the RL approach, which learns the improvement/deterioration of constraints based on rewards, is capable of capturing the relationships between variables and constraints. Since sub-second timing is not reported in Gurobi logs, precise comparisons are challenging. However, all methods found the first feasible solution under 1 second on average. In the D_{9*18} dataset, M_MLP and M_CNN reached RP_Gaps of 4.45% and 1.90%, respectively, while the proposed model and Gurobi achieved 0% RP_Gap by finding the optimal solution. The proposed model not only finds feasible solutions but also reaches the optimal solution.

For the larger D_{100*50} dataset, the proposed model achieved a near-optimal solution with an RP_Gap of 1.33%. The proposed model and its variants are based on a local search strategy, which does not guarantee reaching the global optimum. This limitation likely contributed to the difficulty in finding the optimal solution for the larger search space of D_{100*50}. Nevertheless, the proposed model outperformed M_MLP (1.60%) and M_CNN (1.96%) in terms of RP_Gap. This result suggests that the use of the Transformer encoder enabled the effective learning of complex relationships between variables connected across multiple constraints. While M_CNN outperformed M_MLP in terms of RP_Gap on the D_{9*18} dataset, its performance degraded on the D_{100*50} dataset. Since the order or position of variables in an MILP matrix holds no specific meaning, CNNs' ability to extract local features may not be fully leveraged in larger datasets with many decision variables.

5 Related Work

Bengio, Lodi, and Prouvost have categorized ML techniques for solving CO problems into three groups. The first group, *Learning to Configure Algorithms*, leverages ML to optimize the configuration of specific techniques in traditional MILP solvers. For instance, ML can improve solving speed by deciding whether to execute specific operations, such as decomposition (Kruber, Lübbecke, and Parmentier 2017). The second group, *Machine Learning Alongside Optimization Algorithms*, involves traditional solvers invoking ML repeatedly during the optimization process to support important decisions, such as selecting the next search region (Lodi and Zarpellon 2017). The third group, *End-to-End Learning*, involves ML directly learning and predicting solutions. This group includes ML-based primal heuristics (Gasse et al. 2019; Nair et al. 2020; Yoon 2022; Han et al. 2023; Cantürk et al. 2024), as well as our method. Existing ML-based primal heuristics output only partial solutions and face challenges in handling non-binary integer variables. Moreover, they rely on supervised learning, which requires high costs for collecting labeled training data. In contrast, our RL-based method does not require labeled data. A previous RL-based study on MILP focuses on heuristics for quickly finding the first feasible solution (Qi, Wang, and Shen 2021) and does not address the search for optimal solutions. To the best of our knowledge, our study is the first to propose an ML-based method that incrementally finds better complete feasible solutions.

6 Conclusion

In this study, we proposed a novel RL-based MILP solver capable of generating complete feasible solutions, unlike previous approaches that delegate sub-problems to traditional solvers. We designed an RL system that enables the agent to capture the relationships between decision variables and constraints while learning to solve MILP problems. The reward functions are tailored for two distinct phases: before and after finding the first feasible solution. To effectively capture relationships among decision variables, we proposed a Transformer-based GNN architecture for the RL agent. Our experiments demonstrated that the Transformer-based GNN outperformed other variants in terms of RP_Gap. The trained agent achieved optimal solutions for small datasets and near-optimal solutions with roughly 1% of RP_Gap for larger datasets. This study proposes a novel RL-based MILP solver, which establishes a foundation for ML-driven approaches that eliminate reliance on traditional solvers. This work focused on an MILP problem where all variables are integers while future research can extend the proposed method to handle continuous variables as well. In addition, the local search strategy faces challenges in reaching the global optimal solution due to its restricted search region. To overcome this limitation, further research on search strategies for MILP is necessary. Building on our study as a cornerstone, future work could explore ML approaches that quickly search for high-quality solutions for large-scale problems, where traditional solvers may struggle due to their exponential time complexity.

Acknowledgements

References

Achterberg, T.; Berthold, T.; and Hendel, G. 2012. Rounding and propagation heuristics for mixed integer programming. In *Operations Research Proceedings 2011: Selected Papers of the International Conference on Operations Research (OR 2011), August 30-September 2, 2011, Zurich, Switzerland*, 71–76. Springer.

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

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

Bertsimas, D.; and Tsitsiklis, J. N. 1997. *Introduction to linear optimization*, volume 6. Athena Scientific Belmont, MA.

Cantürk, F.; Varol, T.; Aydoğan, R.; and Özener, O. Ö. 2024. Scalable Primal Heuristics Using Graph Neural Networks for Combinatorial Optimization. *Journal of Artificial Intelligence Research*, 80: 327–376.

Fischetti, M.; Glover, F.; and Lodi, A. 2005. The feasibility pump. *Mathematical Programming*, 104: 91–104.

Gasse, M.; Bowly, S.; Cappart, Q.; Charfreitag, J.; Charlin, L.; Chételat, D.; Chmiela, A.; Dumouchelle, J.; Gleixner, A.; Kazachkov, A. M.; et al. 2022. The machine learning for combinatorial optimization competition (ml4co): Results and insights. In *NeurIPS 2021 competitions and demonstrations track*, 220–231. PMLR.

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

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, 1263–1272. JMLR.org.

Gorishniy, Y.; Rubachev, I.; and Babenko, A. 2022. On embeddings for numerical features in tabular deep learning. *Advances in Neural Information Processing Systems*, 35: 24991–25004.

Han, Q.; Yang, L.; Chen, Q.; Zhou, X.; Zhang, D.; Wang, A.; Sun, R.; and Luo, X. 2023. A GNN-Guided Predict-and-Search Framework for Mixed-Integer Linear Programming. In *The Eleventh International Conference on Learning Representations*.

Hillier, F. S.; and Lieberman, G. J. 2015. *Introduction to operations research*. McGraw-Hill.

Hubbs, C. D.; Li, C.; Sahinidis, N. V.; Grossmann, I. E.; and Wassick, J. M. 2020. A deep reinforcement learning approach for chemical production scheduling. *Computers & Chemical Engineering*, 141: 106982.

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

Kohavi, R.; et al. 1996. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *Kdd*, volume 96, 202–207.

Kostrikov, I. 2018. PyTorch Implementations of Reinforcement Learning Algorithms. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail.

Kruber, M.; Lübbecke, M. E.; and Parmentier, A. 2017. Learning when to use a decomposition. In *International conference on AI and OR techniques in constraint programming for combinatorial optimization problems*, 202–210. Springer.

Kweon, O.; Kim, B.-I.; Lee, G.; Im, H.; Chung, C. Y.; and Lim, O. K. 2024. Parcel delivery network optimization problem considering multiple hubs and consolidation of small-sized parcels. *Computers & Industrial Engineering*, 191: 110113.

Land, A. H.; and Doig, A. G. 2010. *An automatic method for solving discrete programming problems*. Springer.

Li, Q.; Han, Z.; and Wu, X.-M. 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32(1).

Lin, T.; Wang, Y.; Liu, X.; and Qiu, X. 2022. A survey of transformers. *AI open*, 3: 111–132.

Lodi, A.; and Zarpellon, G. 2017. On learning and branching: a survey. *Top*, 25: 207–236.

Min, E.; Chen, R.; Bian, Y.; Xu, T.; Zhao, K.; Huang, W.; Zhao, P.; Huang, J.; Ananiadou, S.; and Rong, Y. 2022. Transformer for graphs: An overview from architecture perspective. *arXiv preprint arXiv:2202.08455*.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous Methods for Deep Reinforcement Learning. In Balcan, M. F.; and Weinberger, K. Q., eds., *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, 1928–1937. New York, New York, USA: PMLR.

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

Pace, R. K.; and Barry, R. 1997. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3): 291–297.

Papageorgiou, D. J.; Nemhauser, G. L.; Sokol, J.; Cheon, M.-S.; and Keha, A. B. 2014. MIRPLib–A library of maritime inventory routing problem instances: Survey, core model, and benchmark results. *European Journal of Operational Research*, 235(2): 350–366.

Qi, M.; Wang, M.; and Shen, Z.-J. 2021. Smart feasibility pump: Reinforcement learning for (mixed) integer programming. *arXiv preprint arXiv:2102.09663*.

Ren, H.; and Gao, W. 2010. A MILP model for integrated plan and evaluation of distributed energy systems. *Applied energy*, 87(3): 1001–1014.

Rong, Y.; Bian, Y.; Xu, T.; Xie, W.; Wei, Y.; Huang, W.; and Huang, J. 2020. Self-supervised graph transformer on large-scale molecular data. *Advances in neural information processing systems*, 33: 12559–12571.

Tomlin, J. A. 1975. *On scaling linear programming problems*. Springer.

Vaswani, A. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

Wu, X.; Chen, Z.; Wang, W. W.; and Jadbabaie, A. 2023. A Non-Asymptotic Analysis of Oversmoothing in Graph Neural Networks. In *The Eleventh International Conference on Learning Representations*.

Wu, Z.; Jain, P.; Wright, M.; Mirhoseini, A.; Gonzalez, J. E.; and Stoica, I. 2021. Representing long-range context for graph neural networks with global attention. *Advances in Neural Information Processing Systems*, 34: 13266–13279.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2019. How Powerful are Graph Neural Networks? In *International Conference on Learning Representations*.

Yoon, T. 2022. Confidence threshold neural diving. *arXiv preprint arXiv:2202.07506*.

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

Zhang, J.; Zhang, H.; Xia, C.; and Sun, L. 2020. Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140*.

Zuo, Y.; Tharmarasa, R.; Jassemi-Zargani, R.; Kashyap, N.; Thiyagalingam, J.; and Kirubarajan, T. T. 2020. MILP formulation for aircraft path planning in persistent surveillance. *IEEE Transactions on Aerospace and Electronic Systems*, 56(5): 3796–3811.