Autoregressive RL Approach for Mixed-Integer Linear Programs

Paul Mingzheng Tang[1], Moses Hong De Lee[1], and
Hoong Chuin Lau[1] (Corresponding Author)

Singapore Management University
School of Computing and Information Systems
80 Stamford Rd
Singapore 178902
hclau@smu.edu.sg

Abstract. Mixed-Integer Linear Programming (MILP) is a widely used method for modeling combinatorial optimization problems. Due to the NP-hard nature of many of these problems, efforts using machine-learning (ML) have been proposed to generate heuristics to speed up solvers, while maintaining optimality. While there is prior work on using graph neural networks (GNNs) to produce high-quality partial solutions, the methods used are non-auto-regressive which model the prediction of variables as conditionally independent due to concerns with solver runtimes. In this paper, we propose a novel auto-regressive reinforcement learning (RL) framework using GNNs which directly optimize for optimality and solver runtimes. Experimental results show our RL method outperforms the benchmark Predict-And-Search (PNS) method on harder real-world problems (55.7% speedup) with time limits and matches performance on easier problems.

Keywords: Graph Neural Networks · ML for OR · Reinforcement Learning.

1 Introduction

Mixed-Integer Linear Programming (MILP) has widespread applications in various industries, from logistics to finance. These problems are combinatorial optimization problems that can be formulated using linear constraints and objective functions. Commercial solvers have been developed to tackle MILPs in general; however, due to scalability issues, there is ample opportunity for data-driven machine learning (ML) methods to learn domain-specific heuristics and solve MILPs more efficiently.

[1, 3] is a primal heuristic that obtains a feasible solution for MILPs by assigning all or a subset of variables. Prior ML-based diving approaches often assign a subset of variables in a non-autoregressive manner (i.e., predicting all variable assignments in parallel). These methods frame the task as learning a probability distribution over variable assignments, which is trained to match the distribution

of assignments sampled from a pool of reference solutions. A key assumption in these approaches is that variable assignments are conditionally independent of one another.

In contrast, this paper proposes a novel autoregressive constructive heuristic for generating partial solutions to MILPs. By framing the task as learning a policy that maximizes a cumulative reward. Incorporating both the optimality gap and a time limit, we train a reinforcement learning (RL) model that directly optimizes the optimality gap within a specified time budget, closely aligning with real-world requirements.

Additionally, our method leverages the additive nature of partial-solution construction by interleaving a variable propagation step into the process. This reduces the number of variables that need to be predicted, distinguishing our approach from existing methods.

The contributions of this paper are as follows: i) a novel constructive heuristic that directly optimizes for a desired optimality gap within a specified time limit; ii) a variable assignment propagation procedure, interleaved into the RL environment, that reduces the number of assigned variables while maintaining solution feasibility.

2 Related Work

2.1 Graph Neural Networks(GNNs) for MILPs

Before selecting an ML method to solve a MILP, the problem must be represented in a form that downstream models can ingest. MILPs are flexible: we can add or remove decision variables and constraints, and both variables and constraints are permutation invariant. Following [4], we represent a MILP as a bipartite graph. This graph can be processed by GNNs to produce predictions at the graph level, as well as at the variable-node and constraint-node levels. A common architecture [4, 11, 6, 9] is the bipartite Graph Convolutional Network (GCN), which uses separate learnable weights for message passing from constraint to variable nodes and from variable to constraint nodes. Our work extends the bipartite graph input representation and utilizes the bipartite GCN as the underlying architecture.

2.2 Reinforcement Learning (RL) for Combinatorial Optimization

There is extensive literature on applying RL to combinatorial optimization, but most approaches target specific problems—such as the Traveling Salesman Problem (TSP) or the Maximum Independent Set (MIS)—and often rely on hand-engineered decoding techniques [2].

RL-MILP [10] is an RL framework that trains a GNN model to learn a local-search heuristic for mixed-integer programming. In contrast, our approach formulates the task as learning a constructive heuristic that directly optimizes both solution quality and solver runtimes.

3 Preliminaries

3.1 MILP Formulation and Terminology

Formally, MILPs can be formulated as shown in Equation 1.

$$\arg\min_{x}\{c^{\top}x | x \in \mathbb{Z}^{q} \times \mathbb{R}^{n-q}, Ax \leq b, l \leq x \leq u\}, \tag{1}$$

where

- x is a solution vector comprising q integer and $(n-q)$ real components,
- $c \in \mathbb{R}^{n}$ is the objective-function coefficient vector,
- $A \in \mathbb{R}^{m \times n}$ is the constraint-coefficient matrix,
- $b \in \mathbb{R}^{m}$ is the right-hand-side (RHS) constant vector,
- $l, u \in \mathbb{R}^{n}$ are the component-wise lower and upper bounds on x.

We adopt the following terminology throughout the paper:

- **LHS**: the left-hand-side linear expression in each constraint (row of Ax).
- **RHS**: the right-hand-side constant term in each constraint (b).
- **Sense**: the inequality symbol ($\leq$, $\geq$, $=$). Although one can standardize a MILP to a single sense (e.g., all constraints as "$\leq$"), we retain each original sense, since different inequalities may require distinct feature-engineering treatments.
- **LB / UB**: the lower and upper bounds on the LHS value given a partial assignment x. We denote these bounds by LB_x and UB_x, respectively.

3.2 Bipartite Graph Representation

A key development in applying ML techniques to MILP problems has been the introduction of graph-based representations. Gasse et al. [4] proposed a lossless representation of MILPs as bipartite graphs. This formulation captures the structure of a MILP in a way that can be efficiently processed by graph neural networks (GNNs).

In this representation, one set of nodes corresponds to the MILP's variables, while the other represents its constraints. Edges connect each variable to the constraints in which it appears, and the edge weights encode the corresponding coefficients from the constraint matrix. This approach enables the encoding of both the problem's structural information and its numerical data.

The bipartite graph representation offers several advantages:

1. It naturally captures variable–constraint interactions.
2. It enables the use of graph-based ML models (e.g., GNNs) that can effectively learn from graph-structured data.
3. It scales to large MILP instances, since the size of the graph grows linearly with the number of variables and constraints.

This graph-based approach has been used in various ML frameworks for MILP solving, including those that predict branching decisions in branch-and-bound algorithms and those that generate initial solutions or guide the search process [4, 11, 9].

3.3 Reinforcement Learning

The fundamental components of RL include the environment, states, actions, and rewards. Let $\mathcal{S}$ denote the set of possible states, $\mathcal{A}$ the set of possible actions, and $\mathcal{R}$ the set of possible rewards. At each time step t, the agent observes the current state $s_t \in \mathcal{S}$, takes an action $a_t \in \mathcal{A}$, and receives a reward $r_t \in \mathcal{R}$ from the environment. The environment then transitions to a new state s_{t+1} according to a transition probability function $P(s_{t+1}|s_t, a_t)$. The agent's behavior is governed by a policy $\pi : \mathcal{S} \to \mathcal{A}$, which maps states to actions. The agent's goal is to learn a policy π that maximizes the expected cumulative reward.

3.4 MDP Formulation

Previous work commonly predicts all decision variables in a non-autoregressive manner, primarily due to concerns over long training times associated with autoregressive models. As a result, these methods explicitly model the probability distribution of variable assignments under the assumption of conditional independence, as shown in Equation 2.

$$p_\theta(x|M) = \prod_{i=1}^{n} p_\theta(x_i|M) \tag{2}$$

As noted in previous work, a key limitation of non-autoregressive prediction is its inability to capture multi-modal distributions. In contrast, our approach employs an autoregressive model that avoids the strong assumption that optimal solutions to an MILP are unimodal. We empirically evaluate the benefits of our method on MILP instances known to exhibit multi-modality.

To formulate an autoregressive model, we build upon an idea proposed in the autoregressive ablation study from [11]. The study introduced a fixed number and order of variables to be predicted at each step and explored three different variable ordering strategies: input order, objective coefficient order, and fractionality order. The formulation is presented in Equation 3.

$$p_\theta(x|M) = \prod_{d=1}^{D} p_\theta(x_d|x_{d-1}, x_{d-2}, ..., x_1; M) \tag{3}$$

Our work proposes to allow the GNN to vary the number and order of variables predicted per step, as well as allow the agent to terminate the partial solution construction process to allow the solver to complete the remaining assignments.

Our approach allows the GNN to vary both the number and order of variables predicted at each step, and enables the agent to terminate partial solution construction early, deferring the remaining assignments to the solver.

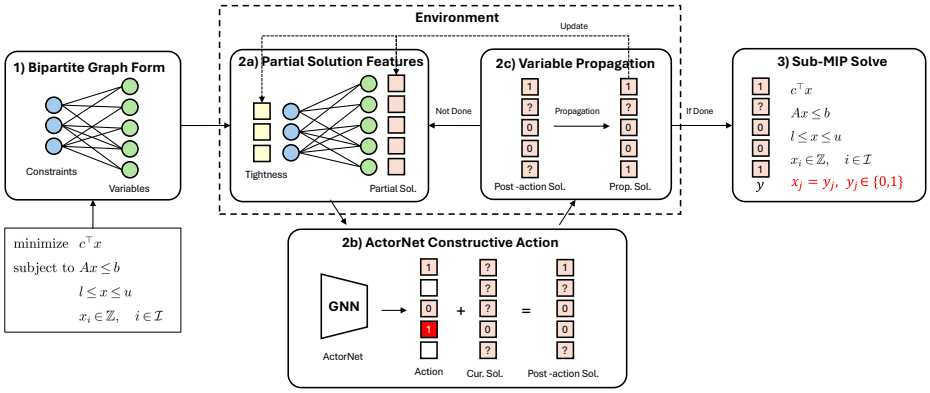


Fig. 1. Overall framework: our approach employs a reinforcement learning–based constructive heuristic that predicts a subset of variable assignments at each step. Once a terminal state is reached, the MILP is solved with the partially assigned variables.

4 Methodology

We propose a novel reinforcement learning framework to solve MILPs.

1. **Bipartite Graph Construction.** We begin by converting the MILP into a bipartite graph, with one node set for variables and another for constraints. Edge weights encode the corresponding entries of the constraint matrix A.
2. **Reinforcement Learning Environment.** The core of our approach unfolds as an episodic interaction between an RL agent and the environment, structured into three sub-steps:
   - **2a)** *Partial-Solution Features.* At each time step, the environment extends the bipartite graph representation with the following: (i) a *tightness metric* for each constraint (i.e., how close the constraint is to binding due to the current partial assignment), and (ii) a *partial solution assignment*, initially all "unknown."
   - **2b)** *ActorNet Constructive Action.* The GNN-based *ActorNet* ingests the current bipartite graph with updated features and outputs a selection and assignment of a subset of variables, producing a *post-action partial solution.*
   - **2c)** *Variable Propagation.* Given the newly assigned variables, our propagation algorithm infers additional assignments implied by the post-action partial solution, yielding a *propagated solution.*This solution—if feasible—is used to update both the tightness metric and the partial solution assignment for the next step.

   The agent repeats steps 2a–2c until a termination criterion is met (maximum number of steps reached or infeasible).
3. **Sub-MIP Solve and Reward Generation.** Once the episode terminates, we are able to fix the assignments according to the partial solution generated using one of two methods:

- *Direct Fixing:* apply the post-processing technique of [11] to complete and fix the assignments;
- *Trust-Region:* invoke the trust-region method of [6] to adjust assignments within a local neighbourhood.

Finally, we solve a small "Sub-MIP" to obtain a feasible full solution and compute the episode reward based on the resulting optimality gap and total solve time.

By iteratively selecting variable subsets and propagating implications, the GNN-based ActorNet learns to capture rich conditional dependencies among variables, directly optimizing for both solution quality and solver runtime.

Proximal Policy Optimization (PPO) In our approach, we used the state-of-the-art RL PPO training method, requiring an Actor Network that is used to select actions in the environment, and a Critic Network, used to estimate the value of the state-action pair taken at that specific state. The model architecture is elaborated further in 4.2.

4.1 Environment Design

State Space. We extend the variable and constraint node feature-engineering used in Predict-And-Search [11] to include the additional information introduced by the partial solution condition. This modified bipartite graph representation defines the observed state space for our RL agent.

To add the partial solution condition into the bipartite graph representation $(\mathcal{G}, \mathcal{C}, \mathcal{V})$, we concatenate the additional feature column to $\mathcal{V}$, where 0 and 1 represents a fixed assignment and 2 represents an unknown assignment.

With the inclusion of a partial solution, we enhance the features for the constraints. In this section, we introduce a new feature, *tightness*, which measures the range of possible values the LHS of the constraint can take.

We define *tightness* as shown in Equation 4

$$\alpha \;=\; \frac{\mathrm{RHS} - \mathrm{LB}_x}{\mathrm{UB}_x - \mathrm{LB}_x},$$

$$T_{c,x} = \begin{cases} \alpha, & \text{if c.Sense is} \geq \\ 1-\alpha, & \text{if c.Sense is} \leq \\ max(\alpha, 1-\alpha), & \text{if c.Sense is} = \end{cases} \tag{4}$$

The *tightness* feature has the following properties:

- When $T_{c,x} = 1$, it implies that the constraint c is *binding*. This allows us to infer the assignment of these variables given a partial solution, and is used in our variable propagation method.
- When $T_{c,x} \leq 0$, it implies that the constraint c is *redundant* and can be relaxed. We also make use of this in our variable propagation method.
- when $T_{c,x} > 1$, there is no feasible assignment of the remaining unfixed variables for this constraint c.

Action Space. The action space for assigning values to n binary variables is defined as:

$$\mathcal{A} = \{0, 1, 2\}^n,$$

where each action vector $\mathbf{a} = (a_1, \ldots, a_n)$ corresponds to one of three possible assignments for variable i:

0/1: Fix variable i to $a_i = 0$ or $a_i = 1$
2: Defer assignment (leaving a_i unresolved)

State Transition. In our RL environment, executing action a_t in state s_t yields the next state s_{t+1} and reward r_{t+1}. The transition proceeds as follows:

1. **Action Filtering.** Mask a_t by the set of variables already fixed in s_t, producing a filtered action a'_t. Represent each new assignment in a'_t as a tuple (v, x) where variable v is assigned a value x for each assignment.
2. **Variable Propagation.** By iteratively assigning each v to the value x, we compute the updated bounds of the constraints' LHS which allow us to find implied assignments to other variables and obtain a larger propagated partial solution. The variable propagation routine is described in (Algorithm 1), which
   - uses the *Propagate-Binding* subroutine (Algorithm 2) on binding constraints, and
   - uses the *Propagate-Redundant* subroutine (Algorithm 3) on redundant constraints,

   to infer any additional assignments or detect infeasibility.

Termination Criterion To prevent the risk of over-assigning variables, which could lead to infeasible solutions, the environment terminates when the number of assigned variables reaches a predetermined proportion. This strategy not only caps excessive assignments but also maintains sufficient flexibility during solve, reducing the likelihood of infeasible outcomes.

Upon triggering the termination condition, the chosen variable assignments are configured and solved using Gurobi. The output from this phase yields metrics that are used in formulating a reward function designed to balance solution quality with solver runtimes.

Reward Computation. The purpose of the reward is to guide the actor to produce actions that match our desired outcomes. There are 2 main objectives we want to optimize for: 1) optimality and feasibility, 2) solver runtimes, which is affected by solver time.

Additionally, to use Proximal Policy Optimization (PPO) efficiently, the rewards produced by the environment should not be sparse. We introduce an intermediate reward (action reward) to avoid a sparse reward landscape.

The reward function is computed in a lexicographic manner, prioritizing in the following order:

Algorithm 1: Propagate Assignment Algorithm

Input : Problem P, Current Solution S, Assignments(Dict) A
Output: isFeasible

```
while A is not empty and isFeasible do
    Variable v, Assignment x = A.pop()
    if S[v] = x then
        continue
    if S[v] = !x then
        return isFeasible = False
    S[v] ← x
    for Constraint C, weight W in v.Constraints do
        Update C.lb and C.ub given v, w, x
        isFeasible = PropagateBinding(S, A, C)
        PropagateRedundant(S, A, C)
        if not isFeasible then
            return isFeasible = False
    #PropagateBinding (PB) and PropagateRedundant (PR) details in
     Algorithms 2 and 3
return isFeasible = True
```

1. Feasibility (r_f)
2. Optimality (r_o)
3. Speed (r_t)

An exploration reward (r_a) and a step penalty (stepP) are given regardless of the 3-stage reward results above. This is formulated as in Equations 5 to 9.

$$r = r_a - \text{stepP} + \begin{cases} r_f, & \text{if infeasible} \\ r_o, & \text{if feasible} \\ r_o + r_t, & \text{if near-optimal} \end{cases} \tag{5}$$

$$r_f = -\frac{\text{numViols}}{\text{numConstrs}} \times \phi_1 \tag{6}$$

$$r_o = (1 - \text{optGap}) \times \phi_2 + \phi_3 \tag{7}$$

$$r_t = \frac{\text{BST} - \text{CST}}{\text{BST}} \times \phi_4 \tag{8}$$

$$r_a = \underbrace{\phi_5 \times \mathbf{1}[\text{num1s} > 0]}_{\text{num 1's reward}} + \underbrace{\phi_6 \times \frac{max(\text{num2s} - \text{solLen}/3, 0)}{2/3 \times \text{solLen}}}_{\text{num 2's reward}} \tag{9}$$

The reward components are as follows:

Algorithm 2: PB Algo

Input : CurrentSol S,
Assignments A,
Constraint C
Output: isFeasible

if *$C.sense$ is $\leq$ and $C.lb(S) = C.rhs$* **then**
  Minimize assignments of $v \in C.vars$ if $S[v]$ unassigned.
if *$C.sense$ is $\geq$ and $C.ub(S) = C.rhs$* **then**
  Maximize assignments of $v \in C.vars$ if $S[v]$ unassigned.
if *new assignments conflict with A or S* **then**
  return isFeasible = False
Push new assignments to A

Algorithm 3: PR Algo

Input : CurrentSol S,
Assignments A,
Constraint C

if *($C.sense$ is $\leq$ and $C.ub(S) \leq C.rhs$) or*
($C.sense$ is $\geq$ and $C.lb(S) \geq C.rhs$) **then**
  for $v \in C.variables$ **do**
    $v.Degree \leftarrow v.Degree - 1$
    if $v.Degree = 0$ **then**
      Assign v which minimizes Objective

Fig. 2. Propagate-Binding (PB) and Propagate-Redundant (PR) algorithms

- r is the **Total Reward**.
- r_a is the **Action Reward**. The action reward functions as an intermediate reward, which provides a positive reward for 2 properties: i) assigning at least one 1; ii) assigning a large number of 2's in the output action. The reason for actions having at least one assignment of 1 is to encourage each action have significant information as problem instance solutions tend to have a sparse number of 1's which are crucial to the optimality of the solution. Additionally, encouraging small actions by providing a positive reward when the number of 2's assigned (deferring the variable assignment) will allow the variable propagation method to mask further actions and provide more information in the next step. Large actions can easily reach infeasible solutions.The choice of normalizing by solLen/3 is to allow the model to learn better solutions from a random actor output (having $\frac{1}{3}$ probability of assigning a 2).
- r_f is the **Feasibility Reward**. The feasibility reward assigns a negative reward when the solution or partial solution is infeasible. It computes the percentage of violated constraints by solving a maximum satisfiable version of the problem instance.
- r_o is the **Optimality Reward**. The optimality reward assigns a positive reward for reducing the optimality gap optGap $= \frac{\text{obj}-\text{BKS}}{|\text{BKS}|}$, scaled by ϕ_2. This includes a fixed reward for obtaining a feasible solution ϕ_3.
- r_t is the **Time reward**. The time reward assigns a positive reward which corresponds to the percentage speed-up obtained from solving the current

sub-MIP, which has a current solve runtime CST, compared to the base solve time BST.

The choice of ϕ scales the various components, which we use $[20, 20, 10, 10, 2, 10]$ for ϕ_1 to ϕ_6 respectively. The step penalty stepP is set to $\frac{1}{\text{BST}}$. Lastly, we clip the reward to ensure more stable training for large negative values to -30. Formally, $r = max(r, -30)$.

4.2 Model Training

We employ two primary networks in our framework: the Actor Network and the Critic Network. Additionally, an Observer network is used to enrich the state features that are passed to both the Actor and Critic Networks.

Network Components

- **Observer Network** builds on the GCN architecture from Predict and Search [6], it is pre-trained via supervised learning for one-step variable solution prediction. As shown in Figure 3, the Observer (OBS GNN) is integrated into both Actor and Critic networks.
- **Actor Network** is a Probabilistic Actor that uses the enriched state features to select actions for the current step. An additional skip connection carries the current solution directly to the layer immediately preceding the output. This connection helps retain critical information from the previous state's partial solution.
- **Critic Network** In parallel, the Critic network aggregates the state features before processing them through its MLP to predict the Q-value of the state-action pair.

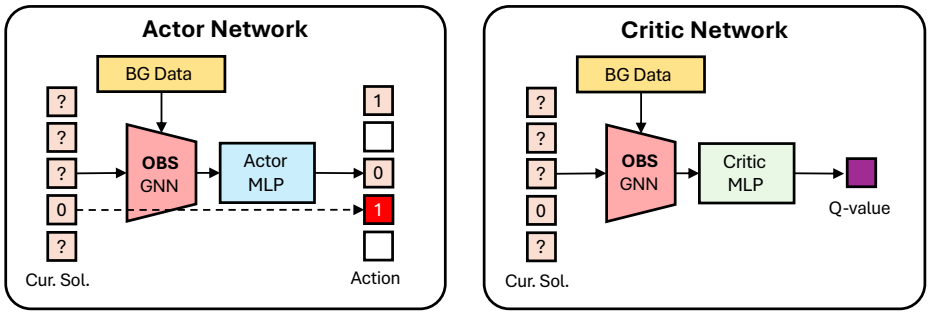


Fig. 3. Actor-Critic Network Architecture: Both the Actor and Critic Network utilize the pre-trained Observer Graph-Neural-Network (OBS GNN) to produce latent graph embeddings of the state features. The Actor's primary goal is to select actions while the critic's role is to evaluate the value of the actions taken within a given state, which are modeled using Multi-Layer Perceptrons (MLP).

Decoding Method We investigate two distinct decoding strategies for configuring the solver:

- **Neural Diving:** This approach fixes variables directly and selects the optimal solution from k candidate samples.
- **Trust Region Method:** Here, the solver is constrained to search within a predefined radius to balance exploration and exploitation [6].

Additionally, we explore the impact of excluding variables assigned to 0 in the predicted solutions. Potentially, predicting only variables to fix as 1 (while omitting 0s) may suffice and even improve solver performance. By restricting assignments to a subset of promising variables, the solver retains flexibility to explore complementary assignments, provided the selected variables (fixed as 1) are high-quality candidates.

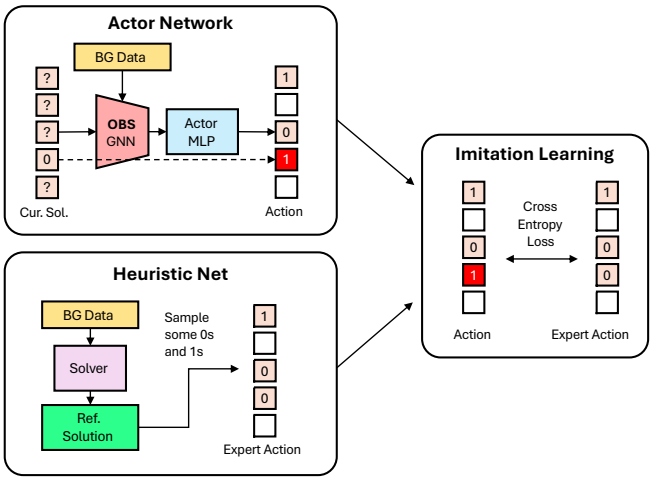


Fig. 4. Expert Heuristic. The expert heuristic iteratively samples a small percentage of 0's and 1's to assign to the current partial solution from a reference solution found in the base solve. Trajectories simulated by this expert heuristic is used to train the actor net using behavior cloning.

Imitation Learning Similar to the imitation learning scheme to learn the strong branching rule for a Branch and Bound GNN model in [4], our work uses behavior cloning to warm-start the actor net using an expert heuristic as shown in Figure 4. The expert heuristic samples actions containing a small fraction of 0's and 1's from a reference solution, and terminates if the size of the partial solution is sufficiently large. Details for the implementation of the expert heuristic is shown in Algorithm 4 in Appendix A.

5 Experiments

5.1 Datasets

We use the following Distributional MIPLIB[8] datasets to evaluate our method, taking 50 instances each (30 for training, 10 for validation, 10 for testing):

1. Set Covering Problem (SC)
2. Combinatorial Auction (CA)
3. Generalized Independent Set Problem (GISP)

We benchmark our RL method against the Predict-and-Search (PNS) approach [6], which trains a one-step, non-autoregressive prediction model using the same trust region decoding strategy.

Preliminaries To reflect real-world scenarios requiring quick solutions, each dataset was solved as a MILP with a 300 seconds time limit, from which the top 100 solutions were extracted. Optimal solutions were found for SC, while CA included some suboptimal ones, and GISP had many.

Set Covering Problem We use the Set Covering Problem as an easy benchmark, focusing on medium-difficulty instances from Distributional MIPLIB. Each instance contains 1000 variables and 1000 constraints, with average solve times under 10 seconds. Although the PNS paper suggests these instances are too easy for the solver, we include them to assess performance on simpler problems.

Combinatorial Auction For comparison with PNS, we evaluate on the CA Medium dataset using the same trust region parameters, applied to our RL method. The key difference is that our approach uses propagation to predict a much smaller subset of variables.

Generalized Independent Set Problem To evaluate on a problem closer to real-world instances—where Gurobi is unable to reach an optimal solution within a 300-second time limit—we use the GISP hard instances. GISP can model forest harvesting optimization problems [7] and presents a challenging task even on relatively small Erdős–Rényi graphs with 150 nodes and an edge probability of 0.3. After 300 seconds, Gurobi reports an optimality gap (between the best bound and best-known solution) ranging from 5% to 20%, providing an opportunity for our method to surpass the BKS.

5.2 Metrics

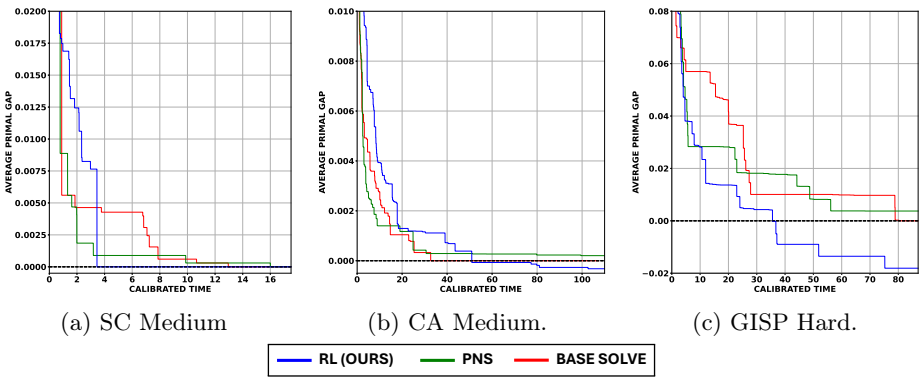
We report the average primal gap and percentage speedup as primary evaluation metrics, following the PNS paper.

Average Primal Gap To compare our RL model with the PNS baseline, we use the average primal gap metric, computed against the best known objective value z^* from the 300 seconds base solve—even if better solutions are later found. To ensure consistency across hardware, solver times are calibrated using a scaling factor, following the Neural Diving paper [11].

Speedup Speedup offers a more intuitive measure, reporting the percentage of time saved by a method relative to the base solve in reaching a solution with objective value at least as good as the base method's best-known solution (BKS).

Evaluation Configurations All evaluations are performed with the same configurations. The hardware we used has a one Intel(R) Core(TM) i7-14700HX @ 2.1Ghz with 20 Cores and 28 logical processors, 64GB of RAM and and one NVIDIA GeForce RTX 4070 GPU. For software, we used Gurobi 12.0.0 Gurobi Optimization, LLC (2025) [5] and pyTorch version 2.5.1. [12]

5.3 Results



(a) SC Medium (b) CA Medium. (c) GISP Hard.

Fig. 5. Average primal gap over calibrated time for SC, CA and GISP datasets. Note that the average primal gap can go below 0 because the BKS objective value used is according to the base solve with a 300s time limit. A negative gap implies a better solution found compared to the base solve. The graphs displayed are of the best decoding methods - see decoding methods ablations in Appendix B.

As shown in Figure 5, our RL-based method outperforms the benchmark PNS approach for all three data sets.

SC Medium Dataset Both methods achieved a 0% optimality gap within the time limit. However, our RL method converged much faster, achieving a

90.5% speedup over the base solver, while PNS showed a more modest 6.5% improvement.

CA Medium Dataset Our method initially took 22.2% longer to match the base solver's best-known value but later found better solutions, yielding a negative average primal gap of 0.04% within 15.34 calibrated seconds. In contrast, PNS reached within 0.02% of the base value but made no further progress.

GISP Hard Dataset Our RL method outperformed both benchmarks, reaching the base solver's best-known value 55.7% faster and achieving a better solution with a negative average primal gap of 1.8% in 75.2 calibrated seconds—faster than the base solver's 80.2 calibrated seconds. PNS failed to reach the base solver's best-known value.

Methodological Considerations The observed under-performance of PNS relative to our base solver, particularly on the combinatorial auction dataset, warrants discussion. Two key factors may explain this discrepancy:

1. **Dataset Scale**: Although we matched the original variable and constraint structure, we trained the PNS model on fewer instances due to computational limitations.
2. **Solver Configuration**: Our experiments used Gurobi's default multi-threaded setting to reflect practical usage, whereas the original PNS results were obtained using single-threaded execution.

Table 1. Average number of predicted ones and zeros and propagated assignments (For RL only) based on the best results shown in Figure 5.

Problem Instance	PNS		RL			
	Predicted		Predicted		Propagated	
	0's	1's	0's	1's	0's	1's
`SC_Medium`	500	0	19.5	1.6	0	0
`CA_Medium`	300	0	15.7	1.2	178.2	0.2
`GISP_Hard`	600	5	143	3.2	706.9	0

Variable Propagation Analysis The figures shown in Table 1 present the average number of predicted ones and zeros, as well as propagated assignments, for a terminal partial solution obtained using our RL method. Compared to PNS,

the results demonstrate how our method reduces the number of predicted variables needed to construct a high-quality partial solution that directly optimizes for solver speedup.

Note that there are no propagated variables for **SC Medium**. The set covering constraints take the form $\sum_{v \in S_i} v \geq 1$, and each variable typically appears in many constraints. Given this, for `PropagateBinding` to be triggered, all but one variable in the constraint must be assigned a value of 0. For `PropagateRedundant` to be triggered, only one variable in the constraint needs to be set to 1. However, this propagation only relaxes the constraint—it does not lead to direct variable assignment unless the degree of the variable is zero. In contrast, the **CA** and **GISP** constraints follow different forms: $\sum_{v \in S_i} v \leq 1$ for CA, and $x_i + x_j \leq 1$ for GISP.

6 Conclusions and Future Work

In conclusion, this paper presents a novel autoregressive RL framework for a constructive MILP partial solution heuristic, which directly optimizes to obtain solutions within an expected optimality gap and specified time limit.

Although our method outperforms the benchmark PNS method and the base Gurobi solver, it should be noted that there are limitations to our framework. A common issue with running RL algorithms is *catastrophic forgetting*, which we currently avoid by terminating model training at its onset. A possible future direction could be to explore other RL techniques to improve the stability of the training.

Furthermore, further investigation of the use of more sophisticated GNN models, such as Graph Transformers[10], could be used to further improve current performance. This paper limits itself to simple MLPs for downstream networks with concerns regarding inference speed and memory usage.

Lastly, there exist more sophisticated propagation methods (consider the presolve methods utilized in commercial solvers that simplify MILPs) that have the potential to propagate more assignments given the same partial solution. Future work could also consider more complex actions, such as repairing infeasible partial solutions, as described in [13].

7 Acknowledgement

This research/project is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-100E-2023-118).

References

1. Berthold, T.: Primal heuristics for mixed integer programs. Ph.D. thesis, Zuse Institute Berlin (ZIB) (2006)

2. Berto, F., Hua, C., Park, J., Luttmann, L., Ma, Y., Bu, F., Wang, J., Ye, H., Kim, M., Choi, S., et al.: Rl4co: an extensive reinforcement learning for combinatorial optimization benchmark. arXiv preprint arXiv:2306.17100 (2023)
3. Eckstein, J., Nediak, M.: Pivot, cut, and dive: a heuristic for 0-1 mixed integer programming. Journal of Heuristics **13**, 471–503 (2007)
4. 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)
5. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2025), https://www.gurobi.com
6. 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. In: The Eleventh International Conference on Learning Representations (2023)
7. Hochbaum, D.S., Pathria, A.: Forest harvesting and minimum cuts: a new approach to handling spatial constraints. Forest Science **43**(4), 544–554 (1997)
8. Huang, W., Huang, T., Ferber, A.M., Dilkina, B.: Distributional miplib: a multi-domain library for advancing ml-guided milp methods. arXiv preprint arXiv:2406.06954 (2024)
9. Khalil, E.B., Morris, C., Lodi, A.: Mip-gnn: A data-driven framework for guiding combinatorial solvers. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 10219–10227 (2022)
10. Lee, T.H., Kim, M.S.: Rl-milp solver: A reinforcement learning approach for solving mixed-integer linear programs with graph neural networks. arXiv preprint arXiv:2411.19517 (2024)
11. 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)
12. Paszke, A.e.a.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019)
13. Salvagnin, D., Roberti, R., Fischetti, M.: A fix-propagate-repair heuristic for mixed integer programming. Mathematical Programming Computation (2024), https://api.semanticscholar.org/CorpusID:259922376

Appendix A Heuristic Expert Algorithm

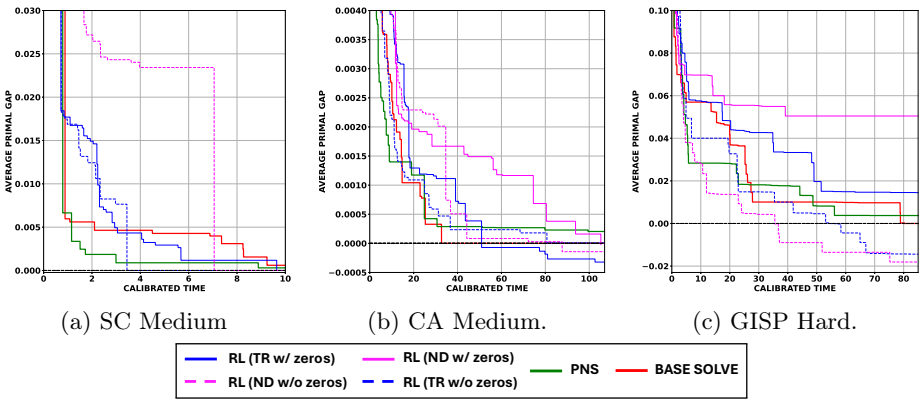
Algorithm 4: HeuristicNet Forward Pass

Input : Partial Solution S, Reference solution R
Output: action

coverage $\leftarrow \frac{\sum(S \neq 2)}{|S|}$
if *coverage* $>$ *coverageThreshold* **then**
  return Terminate action
$k1 \leftarrow \max(1, \lfloor 0.05 \times |S.ones| \rfloor)$
$k0 \leftarrow \max(1, \lfloor 0.005 \times |S.zeros| \rfloor)$
sampledOnes $\leftarrow$ RandomSample($S.ones$, $k1$)
sampledZeros $\leftarrow$ RandomSample($S.zeros$, $k0$)
`#Filter sample 0's and 1's by variables already assigned in` S
action $\leftarrow$ sampledOnes + sampledZeros - S
return action

Appendix B Decoding Methods Ablation

We perform an ablation study exploring the effectiveness of the PNS's trust region decoding against ND's fixed variable decoding, when applied to our RL method. Specifically, the sub-MIP solve uses the trust region constraint $\sum_{v \in X_0} v + \sum_{v' \in X_1} (1 - v') \leq \delta$ instead of the fixed variable constraints $v = 0, \forall v \in X_0$ and $v' = 1, \forall v' \in X_1$, where X_0 and X_1 are the assigned 0's and 1's in the partial solution respectively.



(a) SC Medium (b) CA Medium. (c) GISP Hard.

Fig. 6. Average primal gap over calibrated time for set covering, combinatorial auction and generalized independent set datasets, including the various decoding method configurations. ND and TR refer to Neural Diving decoding and Trust Region decoding respectively. w/ zeros and w/o zeros refer to whether the zeros in the terminal partial solution are kept / removed prior to solving.

We can see from the results in Figure 6 that for SC and CA datasets the PNS method outperforms the ND method. ND for GISP however can perform well when configured to decode without zeros. Across all 3 datasets, ND with zeros (missing in SC diagram because it performs too poorly) does not perform well, which is expected given the lack of flexibility during prediction.