Learning to Search in Local Branching

Defeng Liu [1], **Matteo Fischetti** [2], **Andrea Lodi** [1, 3]

[1]Polytechnique Montréal, [2]University of Padova, [3]Cornell Tech
defeng.liu@polymtl.ca, matteo.fischetti@unipd.it, andrea.lodi@cornell.edu

Abstract

Finding high-quality solutions to mixed-integer linear programming problems (MILPs) is of great importance for many practical applications. In this respect, the refinement heuristic *local branching* (LB) has been proposed to produce improving solutions and has been highly influential for the development of local search methods in MILP. The algorithm iteratively explores a sequence of solution neighborhoods defined by the so-called *local branching constraint*, namely, a linear inequality limiting the distance from a reference solution. For a LB algorithm, the choice of the neighborhood size is critical to performance. Although it was initialized by a conservative value in the original LB scheme, our new observation is that the "best" size is strongly dependent on the particular MILP instance. In this work, we investigate the relation between the size of the search neighborhood and the behavior of the underlying LB algorithm, and we devise a leaning based framework for guiding the neighborhood search of the LB heuristic. The framework consists of a two-phase strategy. For the first phase, a *scaled regression* model is trained to predict the size of the LB neighborhood at the first iteration through a regression task. In the second phase, we leverage reinforcement learning and devise a *reinforced neighborhood search* strategy to dynamically adapt the size at the subsequent iterations. We computationally show that the neighborhood size can indeed be learned, leading to improved performances and that the overall algorithm generalizes well both with respect to the instance size and, remarkably, across instances.

1 Introduction

Mixed-integer linear programming (MILP) is a principal optimization formulation for modeling complex combinatorial problems. The exact solution of a MILP model is generally attempted by a branch-and-bound (or branch-and-cut) (Land et al. 2010) framework. Although state-of-the-art MILP solvers experienced a dramatic performance improvement over the past decades, due to the NP-hardness nature of the problem, the computation load of finding a provable optimal solution for the resulting models can be heavy. In many practical cases, feasible solutions are often required within a very restricted time frame. Hence, one is interested in finding solutions of good quality at the early stage of the computation. In fact, it is also appealing to discover early incumbent solutions in the exact enumerate scheme, which improves the primal bound and reduces the size of the branch-and-bound tree by pruning more nodes (Berthold 2013).

In this respect, the concept of heuristic is well rooted as a principle underlying the search of high-quality solutions. In the literature, a variety of heuristic methods have proven to be remarkably effective, e.g., *local branching* (Fischetti and Lodi 2003), *feasibility pump* (Fischetti et al. 2005), *proximal search* (Fischetti and Monaci 2014), *large neighborhood search* (Gendreau et al. 2010), etc. For more details of these methods, the reader is referred to the surveys (Fischetti and Lodi 2010; Gendreau et al. 2010). In this paper, we focus on local branching, a *refinement heuristic* that iteratively produces improving solutions by exploring suitably predefined solution neighborhoods.

Local branching (LB) was one of the first methods using a generic MILP solver as a black-box tool inside a heuristic framework. Given an initial feasible solution, the method first defines a solution neighborhood through the so-called *local branching constraint*, then explores the resulting subproblem by calling a black-box MILP solver. For a LB algorithm, the choice of neighborhood size is crucial to performance. In the original LB algorithm (Fischetti and Lodi 2003), the size of neighborhood is mostly initialized by a small value, then adjusted in the subsequent iterations. Although these conservative settings have the advantage of yielding a series of easy-to-solve subproblems, each leading to a small progress of the objective, there is still a lot of space for improvement. As discussed in (Fischetti and Monaci 2014), a significantly better performance can be potentially achieved with an ad-hoc tuning of the size of the neighborhood. Our observation also shows that the "best" size is strongly dependent on the particular MILP instance. To illustrate this, the performance of different LB neighborhood size settings for two MILP instances are compared in Figure 1. In principle, it is desirable to have neighborhoods to be relatively small to allow for an efficient computation, but still large enough to be effective for finding improved solutions. Nonetheless, it is reasonable to believe that the size of an ideal neighborhood is correlated with the characteristics of the particular problem instance.

Furthermore, it is worth noting that, in many applications, instances of the same problem are solved repeatedly. Prob-

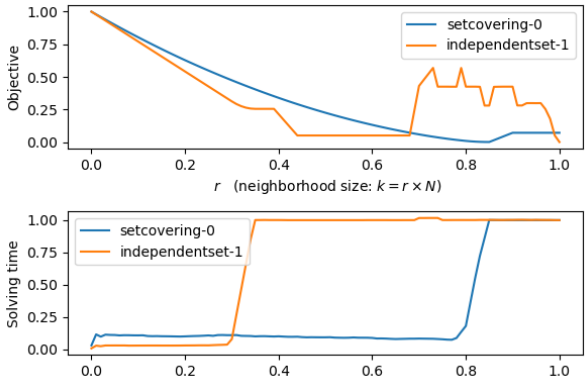

Figure 1: Evaluation of the size of LB neighborhood on a set covering instance and a maximum independent set instance. The neighborhood size k is computed by $k = r \times N$, where N is the number of binary variables, $r \in [0, 1]$. A time limit is imposed for each neighborhood exploration.

lems of real-world applications have a rich structure. While more and more datasets are collected, patterns and regularities appear. Therefore, problem-specific and task-specific knowledge can be learned from data and applied to adapting the corresponding optimization scenario. This motives a broader paradigm of learning to guide the neighborhood search in refinement heuristics.

In this paper, we investigate a learning framework for sizing the search neighborhood of local branching. In particular, given a MILP instance, we exploit patterns in both the problem structure and the statistics of the solving process to predict the size of the LB neighborhood with the aim of maximizing the performance of the underlying LB algorithm. We computationally show that the neighborhood size can indeed be learned, leading to improved performances, and that the overall algorithm generalizes well both with respect to the instance size and, more surprisingly, across instances.

2 Related Work

Recently, the progress in machine learning (ML) has stimulated increasing research interest in learning algorithms for solving MILP problems. These works can be broadly divided into two categories, *learning decision strategies within MILP solvers* and *learning primal heuristics*.

The first approach investigates the use of ML to learn to make decisions inside a MILP solver, which is typically built upon a general branch-and-bound framework. The learned policies can be either cheap approximations of existing expensive methods, or more sophisticated strategies that are new to be discovered. Related works include: learning to select branching variables (Khalil et al. 2016; Balcan 2018; Gasse et al. 2019), learning to select branching nodes (He et al. 2014), learning to select cutting planes (Tang et al. 2020), and learning to optimize the usage of primal heuristics (Khalil et al. 2017; Chmiela et al. 2021).

The *learning primal heuristics* approach is to learn algorithms to produce primal solutions for MILPs. Previous works in this area typically use ML methods to develop *large neighborhood search* (LNS) heuristics. Within an LNS scheme, ML models are trained to predict "promising" solution neighborhoods that are expected to contain high-quality solutions. Ding et al. (2020) trained neural networks to directly predict solution values of binary variables, and then applied the LB heuristic to explore the solution neighborhoods around the predictions. Nair et al. (2020) also used neural networks to predict partial solutions. The subproblems defined by fixing the predicted partial solutions are solved by a MILP solver. Sonnerat et al. (2021) proposed a LNS heuristic based on a "learn to destroy" strategy, which frees part of the current solution. The variables to be freed are selected by trained neural networks using imitation learning. Note that their methods rely on parallel computation, which makes the outcome of the framework within a non-parallel environment less clear. Song et al. (2020) proposed a decomposition-based LNS heuristic. They use imitation learning and reinforcement learning to decompose the set of integer variables into subsets of fixed size. Each subset defines a subproblem. The number of subsets is fixed as a hyperparameter.

Note that the learning-based LNS methods listed above directly operate on the integer variables, i.e., the predictions of ML models are at a variable-wise level, which still encounters the intrinsic combinatorial difficulty of the problem and limits their generalization performances on generic MILPs. Moreover, the learning of these heuristics is mostly based on the extraction of static features of the problem, the dynamic statistics of the heuristic behavior of the solver being barely explored. In this work, we aim to avoid directly making predictions on variables. Instead, we propose to guide the (local) search by learning to control the neighborhood size at an instance-wise level. To identify promising solution neighborhoods, our method exploits not only the static features of the problem, but also the dynamic features collected during the solving process as a sequential approach.

In the literature, there has also been an effort to learn algorithms for solving specific combinatorial optimization problems (Hottung and Tierney 2019; Nazari et al. 2018; Kool et al. 2018; Dai et al. 2017; Bello et al. 2016). For a detailed overview of "learn to optimize", see (Bengio et al. 2021).

3 Preliminaries

3.1 Local Branching

We consider a MILP problem with 0–1 variables of the form

$$
\begin{aligned}
(P) \quad \min \quad & \boldsymbol{c}^T \boldsymbol{x} && (1)\\
\text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, && (2)\\
& x_j \in \{0,1\},\ \forall j \in \mathcal{B}, && (3)\\
& x_j \in \mathbb{Z}^+,\ \forall j \in \mathbb{G}, \quad x_j \geq 0,\ \forall j \in \mathcal{C}, && (4)
\end{aligned}
$$

where the index set of decision variables $\mathcal{N} := \{1, \ldots, n\}$ is partitioned into $\mathcal{B}, \mathcal{G}, \mathcal{C}$, which are the index sets of binary, general integer and continuous variables, respectively.

Note that we assume the existence of binary variables, as one of the basic building blocks of our method—namely, the local branching heuristic—is based on this assumption. However, this constraint can be relaxed and the local branching heuristic can be extended to deal with general integer variables, as proposed in (Bertacco et al. 2007).

Let $\bar{\boldsymbol{x}}$ be a feasible *incumbent* solution for (P), and let $\overline{\mathcal{S}} = \{j \in \mathcal{B} : \bar{x}_j = 1\}$ denote the binary support of $\bar{\boldsymbol{x}}$. For a given positive integer parameter k, we define the neighborhood $N(\bar{\boldsymbol{x}}, k)$ as the set of the feasible solutions of (P) satisfying the *local branching constraint*

$$\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \sum_{j \in \mathcal{B} \setminus \overline{\mathcal{S}}} x_j + \sum_{j \in \overline{\mathcal{S}}} (1 - x_j) \leq k. \tag{5}$$

In the relevant case in which solutions with a small binary support are considered (for example, in the famous traveling salesman problem only n or the $O(n^2)$ variables take value 1), the asymmetric form of local branching constraint is suited, namely

$$\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \sum_{j \in \overline{\mathcal{S}}} (1 - x_j) \leq k. \tag{6}$$

The local branching constraint can be used in an exact branching scheme for (P). Given the incumbent solution $\bar{\boldsymbol{x}}$, the solution space with the current branching node can be partitioned by creating two child nodes as follows:

$$\text{Left: } \Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \leq k, \quad \text{Right: } \Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \geq k + 1.$$

3.2 The Neighborhood Size Optimization Problem

For a *neighborhood size* parameter $k \in \mathbb{Z}^+$, the LB algorithm obtained from choosing k can be denoted as $\mathcal{A}_k$. Given a MILP instance $\boldsymbol{p}$, with its incumbent solution $\bar{\boldsymbol{x}}$, the neighborhood size optimization problem over k for one iteration of $\mathcal{A}_k$ is defined as

$$\min \quad \mathcal{C}(\boldsymbol{p}, \bar{\boldsymbol{x}}; \mathcal{A}_k) \tag{7}$$

$$\text{s.t.} \quad k \in \mathbb{Z}^+, \tag{8}$$

where $\mathcal{C}(\boldsymbol{p}, \bar{\boldsymbol{x}}; \mathcal{A}_k)$ measures the "cost" of $\mathcal{A}_k$ on instance $(\boldsymbol{p}, \bar{\boldsymbol{x}})$ as a trade-off between execution speed and solution quality.

In practice, a run of the LB algorithm consists of a sequence of LB iterations. To maximize the performance of the LB algorithm, a series of the above optimization problems need to be solved. Since the cost function is unknown, those problems cannot be solved analytically. In general, the common strategy is to evaluate some trials of k and select the most performing one. This is often done by using black-box optimization methods (Audet and Hare 2017). As the evaluation of each setting involves a run of $\mathcal{A}_k$ and the best k is instance-specific, those methods are not computationally efficient enough for online use. That is why the original LB algorithm initializes k with a fixed small value and adapts it conservatively by a deterministic strategy.

Currently, learning from experiments and transferring the learned knowledge from solved instances to new instances is of increasing interest and somehow accessible. In the next section, we will introduce a new strategy for selecting the neighborhood size k by using data-driven methods.

4 Methodology

Next, we present our framework for learning the neighborhood size in the LB scheme. The original LB algorithm chooses a conservative value for k as default, with the aim of generating a easy-to-solve subproblem for general MILPs. However, as discussed in Section 1, our observation shows that the "best" k is dependent on the particular MILP instance. Hence, in order to optimize the performance of the LB heuristic, we aim at devising new strategies to learn to tailor the neighborhood size for a specific instance. In particular, we investigate the dependencies between the state of the problem (defined by a set of both static and dynamic features collected during the LB procedure, e.g., context of the problem, incumbent solution, solving status, computation cost, etc.) and the size of the LB neighborhood.

Our framework consists of a two-phase strategy. In the first phase, we define a regression task to learn the neighborhood size for the first LB iteration. Within our method, this size is predicted by a pretrained regression model. For the second phase, we leverage reinforcement learning (RL) (Sutton et al. 1998) and train a policy to dynamically adapt the neighborhood size at the subsequent LB iterations. The exploration of each LB neighborhood is the same as the original LB framework, and a generic MILP solver is used to update the incumbent solution. The overall scheme is exact in nature although turning it into a specialized heuristic framework is trivial (and generally preferrable).

4.1 Scaled Regression for Local Branching

For intermediate LB iterations, the statistics of previous iterations (e.g., value of the previous k, solving statistics, etc.) are available. One can take advantage of this information and exploit the learning methods based on dynamic programming (e.g., reinforcement learning). Section 4.2 will address the case. However, for the first LB iteration, there is no historical information available as input. In this section, we will show how to define a regression task to learn the first k from the context of the problem and the incumbent solution.

Let $\mathcal{S}$ denote the set of available features of the MILPs before the first LB iteration. We aim to train a regression model $f : \mathcal{S} \mapsto \mathbb{R}$ that maps the features of a MILP instance $\boldsymbol{s}$ to k_0^*, the label of best k_0. However, the label k_0^* is unknown, and we do not have any existing method to compute the exact k_0^*. To generate labels, we first define a metric for assessing the performance of a LB algorithm $\mathcal{A}_k$, and then use black-box optimization methods to produce approximations of k_0^* as labels.

Approximation of the Best k_0 To define a cost metric for $\mathcal{A}_k$, we consider two factors. The first factor is the computational effort (e.g., CPU time) to solve the sub-MILP defined by the LB neighborhood, while the second factor is the solution quality (e.g., the objective of the best solution). To quantify the trade-off of speed and quality, the cost metric can be defined as

$$c^{k_0} = \alpha * t_{scaled}^{k_0} + (1 - \alpha) * o_{scaled}^{k_0}, \tag{9}$$

where $t_{scaled}^{k_0} \in [0, 1]$ is the scaled computing time for solving the sub-MILP, $o_{scaled}^{k_0} \in [0, 1]$ is the scaled objective of

sub-MILP, and α is a constant.

Given c^{k_0}, the label k_0^* can be defined as

$$k_0^* = \underset{k_0}{\operatorname{argmin}}\, c^{k_0}, \tag{10}$$

and it is usually evaluated through black-box optimization methods: Given the time limit and a collection of training instances of interest, one evaluates the LB algorithm introduced in Section 3 with different values of k_0, and k_0^* is estimated by choosing the value with the best performance assessed by (9), which is typically the largest k_0 such that the resulting sub-MILP can still be solved to optimality within the time limit. Since the evaluation process for each instance is quite expensive, we propose to approximate it through regression.

Regression for Learning k_0 With a collected dataset $\mathcal{D} = (\boldsymbol{s}_i, k_{0i}^*)_{i=1}^N$ with N instances, a regression task can be analyzed to learn a mapping from the state of the problem to the estimated k_0^*. The regression model $f_\theta(\boldsymbol{s})$ can be obtained by solving

$$\theta^* = \underset{\theta\in\Theta}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f_\theta(\boldsymbol{s}), k_{0i}^*), \tag{11}$$

where $\mathcal{L}(f_\theta(\boldsymbol{s}), k_{0i}^*)$ defines the loss function, a typical choice for regression task being the mean squared error.

The Scaled Regression Task Let $\boldsymbol{x}'$ be the optimal linear programming (LP) fractional solution without local branching constraint, and k' be the value of the left-hand side of the local branching constraint evaluated using $\boldsymbol{x}'$. Specifically, k' is computed by

$$k' = \Delta(\boldsymbol{x}', \bar{\boldsymbol{x}}). \tag{12}$$

As discussed in (Fischetti and Monaci 2014), any $k \geq k'$ is likely to be useless as the LP solution after adding the LB constraint would be unchanged. Hence, k' provides an upper bound for k.

We can therefore parametrize k as

$$k = \phi\, k', \tag{13}$$

where $\phi \in (0, 1)$. Now, we define the regression task over a scaled space $\phi \in (0, 1)$ instead of directly over k.

Given k_0^* and k_0', the label is easily computed by

$$\phi_0^* = \frac{k_0^*}{k_0'}. \tag{14}$$

The regression problem reduces to

$$\theta^* = \underset{\theta\in\Theta}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f_\theta(s), \phi_{0i}^*), \tag{15}$$

where $\mathcal{L}(f_\theta(s), \phi_{0i}^*)$ defines the loss function.

Feature Design We represent the state $\boldsymbol{s}$ as a bipartite graph $(\mathbf{C}, \mathbf{E}, \mathbf{V})$ (Gasse et al. 2019). Given a MILP instance, let n be the number of variables with d features for each variable, m be the number of constraints with q features for each constraint. The variables of the MILP, with $\mathbf{V} \in \mathbb{R}^{n\times d}$ being their feature matrix, are represented on one side of the graph. On the other side are nodes corresponding to the constraints with $\mathbf{C} \in \mathbb{R}^{m\times q}$ being their feature matrix. A constraint node i and a variable node j are connected by an edge (i, j) if variable i appears in constraint j in the MILP model. Finally, $\mathbf{E} \in \mathbb{R}^{m\times n\times e}$ denotes the tensor of edge features, with e being the number of features for each edge.

Algorithm 1: LB with Scaled Regression

Input: instance dataset $\mathcal{P} = \{\boldsymbol{p}_i\}_{i=1}^M$
for *instance* $\boldsymbol{p}_i \in \mathcal{P}$ **do**
  0. initialize the state $\boldsymbol{s}$ with an initial solution $\bar{\boldsymbol{x}}$
  1. solve the LP relaxation and get solution $\boldsymbol{x'}$
  2. compute $k' = \Delta(\boldsymbol{x'}, \bar{\boldsymbol{x}})$
  3. predict $\phi_0 = f_\theta(s)$ by the regression model
  4. compute $k_0 = \phi_0\, k'$
  5. apply k_0 to execute the first LB iteration
  6. update the incumbent $\bar{\boldsymbol{x}}$ and continue LB algorithm with its default setting
  repeat
    execute the next LB iteration
  until *termination condition is reached*;
end
return $\bar{\boldsymbol{x}}$

Regression Model Given that states are represented as graphs, with arbitrary size and topology, we propose to use *graph neural networks* (GNNs) (Gori et al. 2005; Hamilton et al. 2017) to parameterize the regression model. Indeed, GNNs are size-and-order invariant to input data, i.e., they can process graphs of arbitrary size, and the ordering of the input elements is irrelevant. Another appealing property of GNNs is that they exploit the sparsity of the graph, which makes GNNs an efficient model for embedding MILP problems that are typically very sparse (Gasse et al. 2019).

Our GNN architecture consists of 3 modules: the input module, the convolution module, and the output module. In the input layer, the state $\boldsymbol{s}$ is fed into the GNN model. The input module embeds the features of the state $\boldsymbol{s}$. The convolution module propagates the embedded features with graph convolution layers. In particular, our graph convolution layer applies the message passing operator, defined as

$$\mathbf{v}_i^{(h)} = f_\theta^{(h)}\left(\mathbf{v}_i^{(h-1)}, \sum_{j\in\mathcal{N}(i)} g_\phi^{(h)}\left(\mathbf{v}_i^{(h-1)}, \mathbf{v}_j^{(h-1)}, \mathbf{e}_{j,i}\right)\right), \tag{16}$$

where $\mathbf{v}_i^{(h-1)} \in \mathbb{R}^d$ denotes the feature vector of node i from layer $(h-1)$, $\mathbf{e}_{j,i} \in \mathbb{R}^m$ denotes the feature vector of edge (j, i) from node j to node i of layer $(h-1)$, $\mathbf{f}_\theta^{(h)}$ and $\mathbf{g}_\phi^{(h)}$ denote the embedding functions in layer h.

For a bipartite graph, a convolution layer is decomposed into two half-layers: one half-layer propagates messages from variable nodes to constraint nodes through edges, and the other one propagates messages from constraint nodes to variable nodes. The output module embeds the features extracted from the convolution module and then applies a pooling layer, which maps the graph representation into a single neuron. The output of this neuron is the prediction of ϕ_0.

LB with Scaled Regression Our refined LB heuristic, *LB with scaled regression*, is obtained when k_0 is predicted by the regression model. The pseudocode of the algorithm is outlined in Algorithm 1.

4.2 Reinforced Neighborhood Search

In this section, we leverage reinforcement learning (RL) to adapt the neighborhood size iteratively. We first formulate the problem as a Markov Decision Process (MDP) (Howard 1960). Then, we propose to use policy gradient methods to train a policy model.

Markov Decision Process Given a MILP instance with an initial feasible solution, the procedure can be formulated as a MDP, wherein at each step, a neighborhood size is selected by a policy model and applied to run a LB iteration. In principle, the state space $\mathcal{S}$ is the set of all the features of the MILP model and its solving statistics, which is combinatorial and arbitrarily large. To design an efficient RL framework for this problem, we choose a compact set of features from the solving statistics to construct the state. These features characterize the progress of the optimization process and are instance-independent, allowing broader generalization across instances.

For the action space State($\mathcal{A}$), instead of directly selecting a new k, we choose to adapt the value of k of the last LB iteration. The set of possible actions consists of four options

$$\{+k_{step}, 0, -k_{step}; reset\}, \tag{17}$$

where $+k_{step}$ means increasing k by k_{step}, $-k_{step}$ means decreasing k by k_{step}, 0 denotes keeping k without any change and $reset$ means resetting k to a default value. The policy π maps a state to one of the four actions. The step size k_{step} is a hyperparameter of the algorithm.

The compact description of states and actions offers several advantages. First of all, it simplifies the MDP formulation and makes the learning task easier. In addition, it allows the use of simpler function approximators, which is critical for speeding up the learning process. In Section 5, we will show by training a simple linear policy model using the off-the-shelf policy gradient method that the resulting policy can significantly improve the performance of the LB algorithm.

By applying the updated k, the next LB iteration is executed with time limit t_{limit}. Then, the solving sub-MILP statistics are collected to create the next state. In principle, the reward r is formulated according to the outcome of the last LB iteration, e.g., the computing time and the quality of the incumbent solution.

To maximize the objective improvement and minimize the solving time of the LB algorithm, we define the combinatorial reward as

$$r = o_{imp} * (t_{max} - t_{elaps}), \tag{18}$$

where o_{imp} denotes the objective improvement obtained from the last LB iteration, t_{max} is the global time limit of the LB algorithm, and t_{elaps} is the cumulated running time.

The definition above is just one possibility to build a MDP for the LB heuristic. Actually, defining a compact MDP formulation is critical for constructing efficient RL algorithms for this problem.

Learning Strategy For training the policy model, we use the *reinforce* policy gradient method (Sutton et al. 2000), which allows a policy to be learned without any estimate of the value functions.

The refined LB heuristic, *reinforced neighborhood search* is obtained when the neigborhood size k is dynamically adapted by the RL policy. The pseudocode of the algorithm is outlined in Algorithm 2.

Algorithm 2: Reinforced Neighborhood Search

Input: instance dataset $\mathcal{P} = \{\boldsymbol{p}_i\}_{i=1}^{M}$
for *instance* $\boldsymbol{p}_i \in \mathcal{P}$ **do**
  0. initialize the state $\boldsymbol{s}$ with an initial solution $\bar{\boldsymbol{x}}$
  1. compute k_0 by the procedure in Algorithm 1 or set k_0 by a default value.
  2. apply k_0 to execute the first LB iteration
  3. collect the new state $\boldsymbol{s}$ and the incumbent $\bar{\boldsymbol{x}}$
  repeat
    update k by the policy $\pi(\boldsymbol{s})$
    apply k and execute the next LB iteration
    collect the new state $\boldsymbol{s}$ with the incumbent $\bar{\boldsymbol{x}}$
  until *termination condition is reached*;
end
return $\bar{\boldsymbol{x}}$

5 Experiments

In this section, we present the details of our experimental results over five MILP benchmarks. We compare different settings of our approach against the original LB algorithm, using SCIP (Gamrath et al. 2020) as the underlying MILP solver.

5.1 Data Collection

MILP Instances We evaluate on five MILP benchmarks: set covering (SC) (Balas and Ho 1980), maximum independent set (MIS) (Bergman et al. 2016), combinatorial auction (CA) (Leyton-Brown et al. 2000), generalized independent set problem (GISP) (Hochbaum and Pathria 1997; Colombi et al. 2017), and MIPLIB 2017 (Gleixner et al. 2021). The first three benchmarks are used for both training and evaluation. For SC, we use instances with 5000 rows and 2000 columns. For MIS, we use instances on Barabási–Albert random graphs with 1000 nodes. For CA, we use instances with 4000 items and 2000 bids. In addition, to evaluate the generalization performance on larger instances, we also use a larger dataset of instances with doubled size for each benchmark, denoted by LCA, LMIS, LCA. The larger datasets are only used for evaluation.

For GISP, we use the public dataset from (Chmiela et al. 2021). For MIPLIB, we select binary integer linear programming problems from MIPLIB 2017. Instances from GISP and MIPLIB are only used for evaluation.

For each instance, an initial feasible solution is required to run the LB heuristic. We use two initial incumbent solutions: (1) the first solution found by SCIP; (2) an intermediate solution found by SCIP, typically the best solution obtained by SCIP at the end of the root node computation, i.e., before branching.

Data Collection for Regression To collect data for the scaled regression task, one can use black-box optimization methods to produce the label ϕ_0^*. As the search space has only one dimension, we choose to use the grid search method. In particular, given a MILP instance, an initial incumbent $\bar{\boldsymbol{x}}$, the LP solution $\boldsymbol{x}'$, and a time limit for a LB iteration, we evaluate ϕ_0 from $(0, 1)$ with a resolution limit 0.01. For each ϕ_0, we compute the actual neighborhood size by $k_0 = k' \, \phi_0$, where $k' = \Delta(\boldsymbol{x}', \bar{\boldsymbol{x}})$. Then, k_0 is applied to execute an iteration of LB. From all the evaluated ϕ_0, the one with best performance is chosen as a label ϕ_0^*.

The state $\boldsymbol{s}$ consists of context features of the MILP model and the incumbent solution. The state $\boldsymbol{s}$ together with the label k^* construct a valid data point ($\boldsymbol{s}$, k^*).

5.2 Experimental Setup

Datasets For each reference set of SC, MIS and CA problems, we generate a dataset of 200 instances, and split each dataset into training (70%), validation (10%), and test (20%) sets. For larger instances, we generate 40 instances of LSC, LMIS and LCA problems, separately. The GISP dataset contains 35 instances. For MIPLIB, we select 29 binary MILPs that are also evaluated by the original LB heuristic (Fischetti and Lodi 2003).

Training and Evaluation For the regression task, the model learns from the features of the MILP formulation and its incumbent solution. We train the regression model with two scenarios: the first one trains the model on the training set of SC, MIS and CA separately, the other one trains a single model on a mixed dataset of the three training sets. The models trained from the two scenarios are compared on the three test sets. For the RL task, we only use the instance-independent features selected from solving statistics, so the RL policy is only trained on the training set of SC, and evaluated on all the test sets.

To further evaluate the generalization performance with respect to the instance size and the instance type, the RL policy (trained on the SC dataset) and the regression model (trained on the SC, MIS and LCS datasets) are evaluated on GISP and MIPLIB datasets.

Evaluation Metrics We use two measures to compare the performance of different heuristic algorithms. The first indicator is the *primal gap*. Let $\tilde{\boldsymbol{x}}$ be a feasible solution , and $\tilde{\boldsymbol{x}}_{opt}$ be the optimal (or best known) solution. The *primal gap* (in percentage) is defined as

$$\gamma(\tilde{\boldsymbol{x}}) = \frac{|c^T \tilde{\boldsymbol{x}}_{\text{opt}} - c^T \tilde{\boldsymbol{x}}|}{|c^T \tilde{\boldsymbol{x}}_{\text{opt}}|} \times 100,$$

where we assume the denominator is nonzero.

For the second measure, we use the *primal integral* proposed by (Berthold 2013), which takes into account both the quality of solutions and the solving time required to find them. To define the primal integral, we first consider a *primal gap function* $p(t)$ as a function of time, defined as

$$p(t) = \begin{cases} 1, & \text{if no incumbent until time } t, \\ \bar{\gamma}(\tilde{\boldsymbol{x}}(t)), & \text{otherwise,} \end{cases}$$

where $\tilde{\boldsymbol{x}}(t)$ is the incumbent solution at time t, and $\bar{\gamma}(\cdot) \in [0, 1]$ is the *scaled primal gap* defined by

$$\bar{\gamma}(\tilde{x}) = \begin{cases} 0, & \text{if } c^T \tilde{\boldsymbol{x}}_{\text{opt}} = c^T \tilde{\boldsymbol{x}} = 0, \\ 1, & \text{if } c^T \tilde{\boldsymbol{x}}_{\text{opt}} \cdot c^T \tilde{\boldsymbol{x}} < 0, \\ \frac{|c^T \tilde{\boldsymbol{x}}_{\text{opt}} - c^T \tilde{\boldsymbol{x}}|}{\max\{|c^T \tilde{\boldsymbol{x}}_{\text{opt}}|, \, |c^T \tilde{\boldsymbol{x}}|\}}, & \text{otherwise.} \end{cases}$$

Let $t_{\text{max}} > 0$ be the time limit. The primal integral of a run is then defined as

$$P(t_{\text{max}}) = \int_0^{t_{\text{max}}} p(t) \, dt.$$

5.3 Results

Algorithmic Comparisons We perform the evaluations of our framework on the following four settings:

- *lb-sr*: Algorithm 1 with regression model trained by a homogenous dataset of *SC*, *MIS*, *CA*, separately;
- *lb-srm*: Algorithm 1 with regression model trained by a mixed dataset of *SC*, *MIS*, *CA*;
- *lb-rl*: Algorithm 2 with setting k_0 by a default value;
- *lb-srmrl*: Hybrid algorithm using regression from Algorithm 1 (with regression model trained by mixed dataset of *SC*, *MIS*, *CA*) and RL from Algorithm 2.

We use the original local branching algorithm as the baseline. All the algorithms use SCIP as the underlying MILP solver. More details of the experiment environment are in the appendices.

The evaluation results for the basic SC, MIS, CA datasets are shown in Table 1 and Table 2. Our first observation is that all the algorithms of our framework significantly outperform the original LB algorithm. Both the primal integral and the final primal gap of the four LB variants are smaller than those of the baseline, showing improved heuristic behavior. Here, we just highlight that, although the regression model trained by using supervised learning and the policy model trained by RL can be used independently, they benefit from being used together. As a matter of fact, the hybrid algorithm *lb-srmrl* combining both methods achieves a solid further improvement and outperforms the other algorithms for most cases.

We also evaluate the impact of the choice of training set for the regression model. By comparing *lb-sr* and *lb-srm*, we observe that the regression model trained on a mixed dataset of *SC*, *MIS*, *CA* shows performance very close to that of the model trained on a homogeneous dataset. Indeed, the GNN networks we used embed the features of the MILP problem and its incumbent solution. In particular, one significant difference between our method and those of previous works is

that, instead of training a separate model for each class of instances, our method is able to train a single model yielding competitive generalization performances across instances. This is because our models predict the neighborhood size at a instance-wise level, rather than making predictions on variables.

Table 1: Average primal integral for SC, MIS, CA problems

	SC		MIS		CA	
Algo.	first	root	first	root	first	root
lb-base	56.641	4.831	4.986	4.840	7.523	2.777
lb-sr	3.282	1.625	1.575	3.279	5.400	1.712
lb-srm	3.910	1.673	1.473	3.352	5.427	1.839
lb-rl	16.490	3.691	2.179	2.239	3.700	1.857
lb-srmrl	**2.874**	**1.462**	**1.125**	**2.121**	**2.780**	**1.389**

Table 2: Average final primal gap (in percentage) for SC, MIS, CA problems

	SC		MIS		CA	
Algo.	first	root	first	root	first	root
lb-base	1136.74	1.16	0.28	0.20	2.64	1.06
lb-sr	1.21	0.96	0.26	**0.18**	2.12	0.79
lb-srm	1.44	0.96	**0.25**	**0.18**	2.07	0.90
lb-rl	7.05	1.11	0.37	**0.18**	0.49	**0.26**
lb-srmrl	**1.15**	**0.69**	**0.25**	0.23	**0.40**	0.27

Broader Generalization Next, we evaluate the generalization performance with respect to the size and the type of instances. Let us formally restate that the regression model is trained on a randomly mixed dataset of SC, MIS and CA problems, and the RL policy is only trained on the training set of SC. We evaluate the trained models on larger instances (LSC, LMIS, LCA) and new MILP problems (GISP, MIPLIB). The results of evaluation on larger instances are shown in Table 3 and Table 4, whereas the results on GISP and MIPLIB datasets, are shown in Table 5 and Table 6.

Table 3: Average primal integral for LSC, LMIS, LCA problems

	LSC		LMIS		LCA	
Algo.	first	root	first	root	first	root
lb-base	59.306	20.278	23.626	23.951	27.598	11.636
lb-srm	3.065	2.571	4.377	7.704	18.121	5.838
lb-rl	43.876	7.562	5.097	6.097	15.326	7.715
lb-srmrl	**2.424**	**1.904**	**1.653**	**5.011**	**9.047**	**4.796**

Overall, all of our learning-based LB algorithms outperform the baseline, and the hybrid algorithm *lb-srmrl* achieves the best performance on most datasets. These results show that our models, trained on smaller instances, generalize well both with respect to the instance size and, remarkably, across instances.

Table 4: Average final primal gap (in percentage) for LSC, LMIS, LCA problems

	LSC		LMIS		LCA	
Algo.	first	root	first	root	first	root
lb-base	8128.58	5.00	15.04	8.20	21.21	3.57
lb-srm	1.52	1.41	0.21	0.38	13.00	1.16
lb-rl	393.27	3.41	0.30	**0.16**	2.79	0.89
lb-srmrl	**0.72**	**0.52**	**0.19**	**0.16**	**1.57**	**0.20**

Table 5: Average primal integral for GISP and MIPLIB problems

	GISP		MIPLIB	
Algo.	first	root	first	root
lb-base	22.176	18.677	14.984	7.363
lb-srm	15.170	11.059	**10.891**	6.739
lb-rl	18.840	15.751	13.247	6.758
lb-srmrl	**13.786**	**9.591**	11.655	**5.408**

Table 6: Average final primal gap (in percentage) for GISP and MIPLIB problems

	GISP		MIPLIB	
Algo.	first	root	first	root
lb-base	27.21	20.65	2380.27	7.25
lb-srm	14.42	7.89	99.93	**5.22**
lb-rl	19.00	15.55	2338.42	6.03
lb-srmrl	**9.44**	**4.82**	**72.25**	6.22

6 Discussion

In this work, we have looked at the local branching paradigm by using a machine learning lens. We have considered the neighborhood size as a main factor for quantifying high-quality LB neighborhoods. We have presented a learning based framework for predicting and adapting the neighborhood size for the LB heuristic. The framework consists of a two-phase strategy. For the first phase, a *scaled regression* model is trained to predict the size of the LB neighborhood at the first iteration through a regression task. In the second phase, we leverage reinforcement learning and devise a *reinforced neighborhood search* strategy to dynamically adapt the size at the subsequent iterations. We have computationally shown that the neighborhood size can indeed be learned, leading to improved performances and that the overall algorithm generalizes well both with respect to the instance size and, remarkably, across instances. Our framework relies on the availability of an initial solution, thus it can be integrated with other refinement heuristics. For future research, it would be very interesting to design more sophisticated hybrid frameworks that learn to optimize multiple refinement heuristics in a more collaborative way.

7 Acknowledgments

We would like to thank Maxime Gasse for helpful discussions on the project. This work was supported by Canada

Excellence Research Chair in Data Science for Real-Time Decision-Making at Polytechnique Montréal. We are indebted to the anonymous reviewers for their helpful feedback.

References

Audet, C.; and Hare, W. 2017. *Derivative-Free and Black-box Optimization*. Springer.

Balas, E.; and Ho, A. 1980. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial Optimization*, 37–60. Springer.

Balcan, M.-F. o. 2018. Learning to branch. In *International conference on machine learning*, 344–353. PMLR.

Bello, I.; et al. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

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

Bergman, D.; et al. 2016. *Decision diagrams for optimization*, volume 1. Springer.

Bertacco, L.; et al. 2007. A feasibility pump heuristic for general mixed-integer problems. *Discrete Optimization*, 4(1): 63–76.

Berthold, T. 2013. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6): 611–614.

Chmiela, A.; et al. 2021. Learning to Schedule Heuristics in Branch-and-Bound. *arXiv preprint arXiv:2103.10294*.

Colombi, M.; et al. 2017. The generalized independent set problem: Polyhedral analysis and solution approaches. *European Journal of Operational Research*, 260(1): 41–55.

Dai, H.; et al. 2017. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*.

Ding, J.-Y.; et al. 2020. Accelerating Primal Solution Findings for Mixed Integer Programs Based on Solution Prediction. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02): 1452–1459.

Fischetti, M.; and Lodi, A. 2003. Local branching. *Mathematical programming*, 98(1-3): 23–47.

Fischetti, M.; and Lodi, A. 2010. Heuristics in mixed integer programming. *Wiley Encyclopedia of Operations Research and Management Science*.

Fischetti, M.; and Monaci, M. 2014. Proximity search for 0-1 mixed-integer convex programming. *Journal of Heuristics*, 20(6): 709–731.

Fischetti, M.; et al. 2005. The feasibility pump. *Mathematical Programming*, 104(1): 91–104.

Gamrath, G.; et al. 2020. The SCIP Optimization Suite 7.0. Technical report, Optimization Online.

Gasse, M.; et al. 2019. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, 15554–15566.

Gendreau, M.; et al. 2010. *Handbook of metaheuristics*, volume 2. Springer.

Gleixner, A.; et al. 2021. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 1–48.

Gori, M.; et al. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, 729–734. IEEE.

Hamilton, W. L.; et al. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*.

He, H.; et al. 2014. Learning to search in branch and bound algorithms. *Advances in neural information processing systems*, 27: 3293–3301.

Hochbaum, D. S.; and Pathria, A. 1997. Forest harvesting and minimum cuts: a new approach to handling spatial constraints. *Forest Science*, 43(4): 544–554.

Hottung, A.; and Tierney, K. 2019. Neural large neighborhood search for the capacitated vehicle routing problem. *arXiv preprint arXiv:1911.09539*.

Howard, R. A. 1960. Dynamic programming and markov processes.

Khalil, E.; et al. 2016. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

Khalil, E. B.; et al. 2017. Learning to Run Heuristics in Tree Search. In *IJCAI*, 659–666.

Kool, W.; et al. 2018. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*.

Land, A. H.; et al. 2010. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, 105–132. Springer.

Leyton-Brown, K.; et al. 2000. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, 66–76.

Nair, V.; et al. 2020. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*.

Nazari, M.; et al. 2018. Reinforcement learning for solving the vehicle routing problem. *arXiv preprint arXiv:1802.04240*.

Song, J.; et al. 2020. A general large neighborhood search framework for solving integer linear programs. *arXiv preprint arXiv:2004.00422*.

Sonnerat, N.; et al. 2021. Learning a Large Neighborhood Search Algorithm for Mixed Integer Programs. *arXiv preprint arXiv:2107.10201*.

Sutton, R. S.; McAllester, D. A.; Singh, S. P.; and Mansour, Y. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, 1057–1063.

Sutton, R. S.; et al. 1998. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

Tang, Y.; et al. 2020. Reinforcement learning for integer programming: Learning to cut. In *International Conference on Machine Learning*, 9367–9376. PMLR.

A Appendix

A.1 Model Architecture

For the regression task, we applied the GNNs model described in the paper with three modules. For the input module, we applied 2-layer perceptions with the *rectified linear unit* (Relu) activation function to embed the features of nodes. For the convolution module, we used two half-layers, one from nodes of variables to nodes of constraints, and the other one from nodes of constraints to nodes of variables. For the output module, we applied 2-layer perceptions with the Relu activation function. The pooling layer uses the *sigmoid* activation function. All the hidden layers have 64 neurons. For the bipartite graph representation, we referenced the model used in Gasse et al. (2019). The features in the bipartite graph are listed in Table 7. In practice, if the instance is a pure binary MILP, one can choose a more compact set of features to accelerate the training process (for example, the features describing the type and bound of the variables can be removed).

Table 7: Description of the features in the bipartite graph $\boldsymbol{s} = (\mathbf{C}, \mathbf{E}, \mathbf{V})$.

Tensor	Feature	Description
$\mathbf{C}$	bias	Bias value, normalized with constraint coefficients.
$\mathbf{E}$	coef	Constraint coefficient, normalized per constraint.
$\mathbf{V}$	coef	Objective coefficient, normalized.
	binary	Binary type binary indicator.
	integer	Integer type indicator.
	imp_integer	Implicit integer type indicator
	continuous	continuous type indicator.
	has_lb	Lower bound indicator.
	has_ub	Upper bound indicator.
	lb	Lower bound.
	ub	Upper bound.
	sol_val	Solution value.

For the RL policy, we use a linear model with seven inputs and four outputs. The feature used by the RL policy is listed in Table 8.

Table 8: Description of the input features of the RL policy.

Feature	Description
optimal	Indicating the subproblem is solved to optimal and an improving solution is found.
infeasible	Indicating the subproblem is proven infeasible.
improved	Indicating the subproblem is not solved but an improving solution is found
not_improved	Indicating the subproblem is not solved and no improving solution is found.
diverse	no improving solution is found for two consecutive iterations.
t_available	time available before the time limit of the subproblem is reached
obj_improve	improvement of objective.