

---

# Exact Combinatorial Optimization with Graph Convolutional Neural Networks

---

**Maxime Gasse**

Mila, Polytechnique Montréal  
maxime.gasse@polymtl.ca

**Didier Chételat**

Polytechnique Montréal  
didier.chetelat@polymtl.ca

**Nicola Ferroni**

University of Bologna  
nicola.ferroni2@studio.unibo.it

**Laurent Charlin**

Mila, HEC Montréal  
laurent.charlin@hec.ca

**Andrea Lodi**

Mila, Polytechnique Montréal  
andrea.lodi@polymtl.ca

## Abstract

Combinatorial optimization problems are typically tackled by the branch-and-bound paradigm. We propose a new graph convolutional neural network model for learning branch-and-bound variable selection policies, which leverages the natural variable-constraint bipartite graph representation of mixed-integer linear programs. We train our model via imitation learning from the strong branching expert rule, and demonstrate on a series of hard problems that our approach produces policies that improve upon state-of-the-art machine-learning methods for branching and generalize to instances significantly larger than seen during training. Moreover, we improve for the first time over expert-designed branching rules implemented in a state-of-the-art solver on large problems. Code for reproducing all the experiments can be found at <https://github.com/ds4dm/learn2branch>.

## 1 Introduction

Combinatorial optimization aims to find optimal configurations in discrete spaces where exhaustive enumeration is intractable. It has applications in fields as diverse as electronics, transportation, management, retail, and manufacturing [41], but also in machine learning, such as in structured prediction and maximum a posteriori inference [48; 32; 46]. Such problems can be extremely difficult to solve, and in fact most classical NP-hard computer science problems are examples of combinatorial optimization. Nonetheless, there exists a broad range of exact combinatorial optimization algorithms, which are guaranteed to find an optimal solution despite a worst-case exponential time complexity [49]. An important property of such algorithms is that, when interrupted before termination, they can usually provide an intermediate solution along with an optimality bound, which can be a valuable information in theory and in practice. For example, after one hour of computation, an exact algorithm may give the guarantee that the best solution found so far lies within 2% of the optimum, even without knowing what the actual optimum is. This quality makes exact methods appealing and practical, and as such they constitute the core of modern commercial solvers.

In practice, most combinatorial optimization problems can be formulated as mixed-integer linear programs (MILPs), in which case branch-and-bound (B&B) [33] is the exact method of choice. Branch-and-bound recursively partitions the solution space into a search tree, and computes relax-

ation bounds along the way to prune subtrees that provably cannot contain an optimal solution. This iterative process requires sequential decision-making, such as *node selection*: selecting the next node to evaluate, and *variable selection*: selecting the variable by which to partition the node’s search space. This decision process traditionally follows a series of hard-coded heuristics, carefully designed by experts to minimize the average solving time on a representative set of MILP instances [40]. However, in many contexts it is common to repeatedly solve similar combinatorial optimization problems, e.g., day-to-day production planning and lot-sizing problems [43], which may significantly differ from the set of instances on which B&B algorithms are typically evaluated. It is then appealing to use statistical learning for tuning B&B algorithms automatically for a desired class of problems. However, this line of work raises two challenges. First, it is not obvious how to encode the state of a MILP B&B decision process [4], especially since both search trees and integer linear programs can have a variable structure and size. Second, it is not clear how to formulate a model architecture that leads to rules which can generalize, at least to similar instances but also ideally to instances larger than seen during training.

In this work we propose to address the above challenges by using graph convolutional neural networks. More precisely, we focus on variable selection, also known as the *branching problem*, which lies at the core of the B&B paradigm yet is still not well theoretically understood [39], and adopt an imitation learning strategy to learn a fast approximation of *strong branching*, a high-quality but expensive branching rule. While such an idea is not new [29; 4; 23], we propose to address the learning problem in a novel way, through two contributions. First, we propose to encode the branching policies into a graph convolutional neural network, which allows us to exploit the natural bipartite graph representation of MILP problems, thereby reducing the need for manual feature engineering. Second, we approximate strong branching decisions by using behavioral cloning with a cross-entropy loss, a less difficult task than predicting strong branching scores [4] or rankings [29; 23]. We evaluate our approach on three classes of NP-hard problems, namely set covering, combinatorial auction, and capacitated facility location. We compare against the previously proposed approaches of Khalil et al. [29], Alvarez et al. [4] and Hansknecht et al. [23] which rely on handcrafted variable features, as well as against the default hybrid branching rule in SCIP [20], a modern open-source solver. The results show that our choice of model, state encoding, and training procedure leads to policies that can offer a substantial improvement over traditional branching rules, and generalize well to larger instances than those used in training.

In Section 2, we review the broader literature of works using statistical learning for branching. In Section 3, we formally introduce the B&B framework, and formulate the branching problem as a Markov decision process. In Section 4, we present our state representation, model, and training procedure for addressing the branching problem. Finally, we discuss experimental results in Section 5.

## 2 Related work

First steps towards statistical learning of branching rules in B&B were taken by Khalil et al. [29], who learn a branching rule customized to a single instance during the B&B process, as well as Alvarez et al. [4] and Hansknecht et al. [23] who learn offline on a collection of similar instances, in a fashion similar to us. In each case a branching policy is learned by imitation of the strong branching expert, although the learning problem is formulated differently. Namely, Khalil et al. [29] and Hansknecht et al. [23] treat it as a ranking problem and learn the partial ordering of the candidates produced by the expert, while, Alvarez et al. [4] treat it as a regression problem and learn directly the strong branching scores of the candidates. In contrast, we treat it as a classification problem and simply learn from the expert decisions, which allows imitation from experts that don’t rely on branching scores or orderings. These works also differ from ours in three other key aspects. First, they rely on extensive feature engineering, which is reduced by our graph convolutional neural network approach. Second, they do not evaluate generalization ability to instances larger than seen during training, which we propose to do. Finally, in each case performance was evaluated on a simplified solver, whereas we compare, for the first time and favorably, against a full-fledged solver with primal heuristics, cuts and presolving. We compare against these approaches in Section 5.

Other works also consider using machine learning to improve variable selection in branch-and-bound, without directly learning a branching policy. Di Liberto et al. [14] learn a clustering-based classifier to pick a variable selection rule at every branching decision up to a certain depth, while Balcan et al. [8] use the fact that many variable selection rules in B&B explicitly score the candidates,

and propose to learn a weighting of different existing scores to combine their strengths. Other works learn variable selection policies, but for less general algorithms than B&B. Liang et al. [37] learn a variable selection policy for SAT solvers using a bandit approach, and Lederman et al. [34] extend their work by taking a reinforcement learning approach with graph convolutional neural networks. Unlike our approach, these works are restricted to conflict-driven clause learning methods in SAT solvers, and cannot be readily extended to B&B methods for arbitrary combinatorial optimization problems. In the same vein, Balunovic et al. [9] learn by imitation learning a variable selection procedure for SMT solvers that exploits specific aspects of this type of solver.

Finally, researchers have also focused on learning other aspects of B&B algorithms than variable selection. He et al. [24] learn a node selection heuristic by imitation learning of the oracle procedure that expands the node whose feasible set contains the optimal solution, while Khalil et al. [30] learn primal heuristics for B&B algorithms. Those approaches are complementary with our work, and could in principle be combined to further improve solver performance. More generally, many authors have proposed machine learning approaches to fine-tune exact optimization algorithms, not necessarily for MILPs in general. A recent survey is provided by Bengio et al. [10].

### 3 Background

#### 3.1 Problem definition

A mixed-integer linear program is an optimization problem of the form

$$\arg \min_{\mathbf{x}} \{ \mathbf{c}^\top \mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \}, \quad (1)$$

where  $\mathbf{c} \in \mathbb{R}^n$  is called the objective coefficient vector,  $\mathbf{A} \in \mathbb{R}^{m \times n}$  the constraint coefficient matrix,  $\mathbf{b} \in \mathbb{R}^m$  the constraint right-hand-side vector,  $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$  respectively the lower and upper variable bound vectors, and  $p \leq n$  the number of integer variables. Under this representation, the size of a MILP is typically measured by the number of rows ( $m$ ) and columns ( $n$ ) of the constraint matrix. By relaxing the integrality constraint, one obtains a continuous linear program (LP) whose solution provides a lower bound to (1), and can be solved efficiently using, for example, the simplex algorithm. If a solution to the LP relaxation respects the original integrality constraint, then it is also a solution to (1). If not, then one may decompose the LP relaxation into two sub-problems, by splitting the feasible region according to a variable that does not respect integrality in the current LP solution  $\mathbf{x}^*$ ,

$$x_i \leq \lfloor x_i^* \rfloor \vee x_i \geq \lceil x_i^* \rceil, \quad \exists i \leq p \mid x_i^* \notin \mathbb{Z}, \quad (2)$$

where  $\lfloor \cdot \rfloor$  and  $\lceil \cdot \rceil$  respectively denote the floor and ceil functions. In practice, the two sub-problems will only differ from the parent LP in the variable bounds for  $x_i$ , which get updated to  $u_i = \lfloor x_i^* \rfloor$  in the left child and  $l_i = \lceil x_i^* \rceil$  in the right child.

The branch-and-bound algorithm [49, Ch. II.4], in its simplest formulation, repeatedly performs this binary decomposition, giving rise to a search tree. By design, the best LP solution in the leaf nodes of the tree provides a lower bound to the original MILP, whereas the best integral LP solution (if any) provides an upper bound. The solving process stops whenever both the upper and lower bounds are equal or when the feasible regions do not decompose anymore, thereby providing a certificate of optimality or infeasibility, respectively.

#### 3.2 Branching rules

A key step in the B&B algorithm is **selecting a fractional variable to branch on in (2)**, which can have a very significant impact on the size of the resulting search tree [2]. As such, branching rules are at the core of modern combinatorial optimization solvers, and have been the focus of extensive research [38; 42; 1; 16]. So far, the branching strategy consistently resulting in the smallest B&B trees is *strong branching* [5]. It does so by computing the expected bound improvement for each candidate variable before branching, which unfortunately requires the solution of two LPs for every candidate. In practice, running strong branching at every node is prohibitive, and modern B&B solvers instead rely on *hybrid branching* [3; 1] which computes strong branching scores only at the beginning of the solving process and gradually switches to simpler heuristics such as: the conflict score (in the original article), the pseudo-cost [42] or a hand-crafted combination of the two. For a

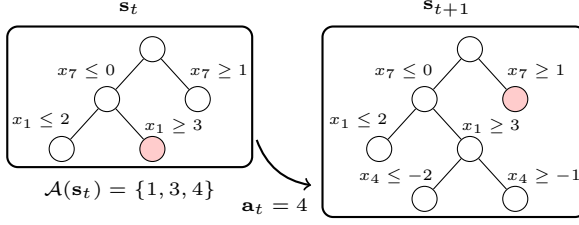


Figure 1: B&B variable selection as a Markov decision process. On the left, a state  $s_t$  comprised of the branch-and-bound tree, with a leaf node chosen by the solver to be expanded next (in pink). On the right, a new state  $s_{t+1}$  resulting from branching on the variable  $a_t = x_4$ .

more extensive discussion of B&B branching strategies in MILP, the reader is referred to Achterberg et al. [3].

### 3.3 Markov decision process formulation

As remarked by He et al. [24], the sequential decisions made during B&B can be assimilated to a Markov decision process [25]. Consider the solver to be the environment, and the brancher the agent. At the  $t^{\text{th}}$  decision the solver is in a state  $s_t$ , which comprises the B&B tree with all past branching decisions, the best integer solution found so far, the LP solution of each node, the currently focused leaf node, as well as any other solver statistics (such as, for example, the number of times every primal heuristic has been called). The brancher then selects a variable  $a_t$  among all fractional variables  $\mathcal{A}(s_t) \subseteq \{1, \dots, p\}$  at the currently focused node, according to a policy  $\pi(a_t | s_t)$ . The solver in turn extends the B&B tree, solves the two child LP relaxations, runs any internal heuristic, prunes the tree if warranted, and finally selects the next leaf node to split. We are then in a new state  $s_{t+1}$ , and the brancher is called again to take the next branching decision. This process, illustrated in Figure 1, continues until the instance is solved, i.e., until there are no leaf node left for branching.

As a Markov decision process, B&B is episodic, where each episode amounts to solving a MILP instance. Initial states correspond to an instance being sampled among a group of interest, while final states mark the end of the optimization process. The probability of a trajectory  $\tau = (s_0, \dots, s_T) \in \mathcal{T}$  then depends on both the branching policy  $\pi$  and the remaining components of the solver,

$$p_\pi(\tau) = p(s_0) \prod_{t=0}^{T-1} \sum_{a \in \mathcal{A}(s_t)} \pi(a | s_t) p(s_{t+1} | s_t, a).$$

In practice, one might look for policies which minimize running time, but other measures that are less hardware-dependent could also be of interest, such as the upper bound integral for finding integer solutions quickly, the negative lower bound integral for tightening the MILP relaxation quickly, or the duality gap integral, which is frequently used in the combinatorial optimization community to measure the quality of a solver. Such measures can be formulated as expected long-term returns with a suitable reward function  $r : \mathcal{T} \rightarrow \mathbb{R}$ , in which case branching is equivalent to the control problem

$$\arg \max_{\pi} \mathbb{E}_{\tau \sim p_\pi} [r(\tau)]. \quad (3)$$

A natural approach to solve this problem is reinforcement learning. However, this raises several key issues which we circumvent by adopting an imitation learning scheme, as discussed next.

## 4 Methodology

We now describe our approach for tackling the B&B variable selection problem in MILPs, where we use imitation learning and a dedicated graph convolutional neural network model. As the B&B variable selection problem can be formulated as a Markov decision process, a natural way of training a policy would be reinforcement learning [47]. However, this approach runs into many issues. Notably, as episode length is proportional to performance, and randomly initialized policies perform poorly, standard reinforcement learning algorithms are usually so slow early in training as to make total training time prohibitively long. Moreover, once the initial state corresponding to an instance is selected, the rest of the process is instance-specific, and so the Markov decision processes tend to be extremely large. In this work we choose instead to learn directly from an expert branching rule, an approach usually referred to as imitation learning [26].

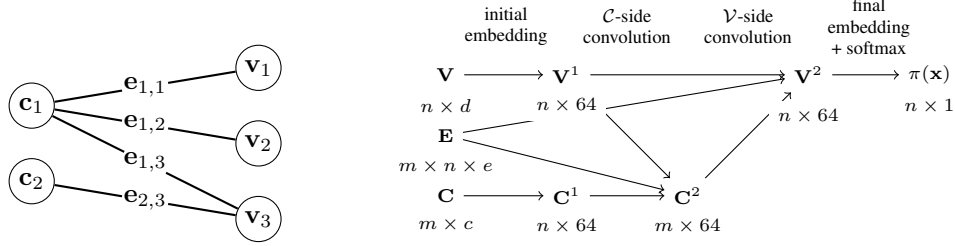


Figure 2: Left: our bipartite state representation  $s_t = (\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$  with  $n = 3$  variables and  $m = 2$  constraints. Right: our bipartite GCNN architecture for parametrizing our policy  $\pi_\theta(\mathbf{a} \mid s_t)$ .

#### 4.1 Imitation learning

We train by behavioral cloning [44] using the strong branching rule, which suffers a high computational cost but usually produces the smallest B&B trees, as mentioned in Section 3.2. We first run the expert on a collection of training instances of interest, record a dataset of expert state-action pairs  $\mathcal{D} = (s_i, \mathbf{a}_i^*)_{i=1}^N$ , and then learn our policy by minimizing the cross-entropy loss

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathbf{s}, \mathbf{a}^*) \in \mathcal{D}} \log \pi_\theta(\mathbf{a}^* \mid \mathbf{s}). \quad (4)$$

#### 4.2 State encoding

We encode the state  $s_t$  of the B&B process at time  $t$  as a bipartite graph with node and edge features  $(\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$ , described in Figure 2 (Left). On one side of the graph are nodes corresponding to the constraints in the MILP, one per row in the current node’s LP relaxation, with  $\mathbf{C} \in \mathbb{R}^{m \times c}$  their feature matrix. On the other side are nodes corresponding to the variables in the MILP, one per LP column, with  $\mathbf{V} \in \mathbb{R}^{n \times d}$  their feature matrix. An edge  $(i, j) \in \mathcal{E}$  connects a constraint node  $i$  and a variable node  $j$  if the latter is involved in the former, that is if  $\mathbf{A}_{ij} \neq 0$ , and  $\mathbf{E} \in \mathbb{R}^{m \times n \times e}$  represents the (sparse) tensor of edge features. Note that under mere restrictions in the B&B solver (namely, by enabling cuts only at the root node), the graph structure is the same for all LPs in the B&B tree, which reduces the cost of feature extraction. The exact features attached to the graph are described in the supplementary materials. We note that this is really only a subset of the solver state, which technically turns the process into a partially-observable Markov decision process [6], but also that excellent variable selection policies such as strong branching are able to do well despite relying only on a subset of the solver state as well.

#### 4.3 Policy parametrization

We parametrize our variable selection policy  $\pi_\theta(\mathbf{a} \mid s_t)$  as a graph convolutional neural network (GCNN) [22; 45; 11]. Such models, also known as message-passing neural networks [19], are extensions of convolutional neural networks from grid-structured data (as in images or sounds) to arbitrary graphs. They have been successfully applied to a variety of machine learning tasks with graph-structured inputs, such as prediction of molecular properties [15; 18], program verification [36], and document classification in citation networks [31]. Graph convolutions exhibit many properties which make them a natural choice for graph-structured data in general, and MILP problems in particular: 1) they are well-defined no matter the input graph size; 2) their computational complexity is directly related to the density of the graph, which makes it an ideal choice for processing typically sparse MILP problems; and 3) they are permutation-invariant, that is they will always produce the same output no matter the order in which the nodes are presented.

Our model takes as input our bipartite state representation  $s_t = (\mathcal{G}, \mathbf{C}, \mathbf{V}, \mathbf{E})$  and performs a single graph convolution, in the form of two interleaved half-convolutions. In detail, because of the bipartite structure of the input graph, our graph convolution can be broken down into two successive passes, one from variable to constraints and one from constraints to variables. These passes take the

Table 1: Imitation learning accuracy on the test sets.

model	Set Covering			Combinatorial Auction			Capacitated Facility Location		
	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10
TREES	49.9 $\pm$ 0.8	78.3 $\pm$ 0.5	89.8 $\pm$ 0.3	51.5 $\pm$ 0.7	82.9 $\pm$ 0.3	93.3 $\pm$ 0.1	59.9 $\pm$ 1.0	95.5 $\pm$ 0.9	98.7 $\pm$ 1.0
SVMRANK	57.3 $\pm$ 0.3	84.6 $\pm$ 0.0	93.9 $\pm$ 0.0	56.8 $\pm$ 0.3	86.3 $\pm$ 0.2	95.1 $\pm$ 0.2	65.9 $\pm$ 0.1	97.4 $\pm$ 0.0	99.8 $\pm$ 0.0
LMART	57.2 $\pm$ 0.1	84.4 $\pm$ 0.1	93.5 $\pm$ 0.1	57.0 $\pm$ 0.4	86.6 $\pm$ 0.3	95.2 $\pm$ 0.2	66.0 $\pm$ 0.2	97.5 $\pm$ 0.0	99.8 $\pm$ 0.0
GCNN	<b>65.6</b> $\pm$ 0.1	<b>92.5</b> $\pm$ 0.1	<b>98.3</b> $\pm$ 0.1	<b>60.8</b> $\pm$ 0.2	<b>90.8</b> $\pm$ 0.1	<b>97.6</b> $\pm$ 0.0	<b>68.9</b> $\pm$ 0.2	<b>98.0</b> $\pm$ 0.1	<b>99.9</b> $\pm$ 0.0

form

$$\mathbf{c}_i \leftarrow \mathbf{f}_C \left( \mathbf{c}_i, \sum_j^{(i,j) \in \mathcal{E}} \mathbf{g}_C(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}) \right), \quad \mathbf{v}_j \leftarrow \mathbf{f}_V \left( \mathbf{v}_j, \sum_i^{(i,j) \in \mathcal{E}} \mathbf{g}_V(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}) \right) \quad (5)$$

for all  $i \in \mathcal{C}, j \in \mathcal{V}$ , where  $\mathbf{f}_C, \mathbf{f}_V$  and  $\mathbf{g}_C, \mathbf{g}_V$  are respectively single-layer and two-layer perceptrons with *relu* activation functions. Following this graph-convolution layer, we obtain a bipartite graph with the same topology as the input, but with potentially different node features, so that each node now contains information from its neighbors. We obtain our policy by discarding the constraint nodes and applying a final 2-layer perceptron on variable nodes, combined with a masked softmax activation to produce a probability distribution over the candidate branching variables (i.e., the non-fixed LP variables). The right side of Figure 2 provides an overview of our architecture.

**Prenorm layers** In the literature of GCNN, it is common to normalize each convolution operation by the number of neighbours [31]. As noted by Xu et al. [50] this might result in a loss of expressiveness, as the model then becomes unable to perform a simple counting operation (e.g., in how many constraints does a variable appears). Therefore we opt for un-normalized convolutions. However, this introduces a weight initialization issue. Indeed, weight initialization in standard CNNs relies on the number of input units to normalize the initial weights [21], which in a GCNN is unknown beforehand and depends on the dataset. To overcome this issue and stabilize the learning procedure, we adopt a simple affine transformation  $\mathbf{x} \leftarrow (\mathbf{x} - \beta)/\sigma$ , which we call a *prenorm* layer, applied right after the summation in (5). The  $\beta$  and  $\sigma$  parameters are initialized with respectively the empirical mean and standard deviation of  $\mathbf{x}$  on the training dataset, and fixed once and for all before the actual training. Adopting both un-normalized convolutions and this pre-training procedure improves our generalization performance on larger problems, as will be shown in Section 5.

## 5 Experiments

We now present a comparative experiment against three competing machine learning approaches and SCIP’s default branching rule to assess the value of our approach, as well as an ablation study to validate our architectural choices. Code for reproducing all the experiments can be found at <https://github.com/ds4dm/learn2branch>.

### 5.1 Setup

**Benchmarks** We evaluate our approach on three NP-hard problem benchmarks. Our first benchmark is comprised of set covering instances generated following Balas and Ho [7], with 1,000 columns. We train and test on instances with 500 rows, and we evaluate on instances with 500 (Easy), 1,000 (Medium) and 2,000 (Hard) rows. Our second benchmark is comprised of combinatorial auction instances, generated following the *arbitrary relationships* procedure of Leyton-Brown et al. [35, Section 4.3]. We train and test on instances with 100 items for 500 bids, and we evaluate on instances with 100 items for 500 bids (Easy), 200 items for 1,000 bids (Medium) and 300 items for 1,500 bids (Hard). Finally, our third benchmark is comprised of capacitated facility location instances generated following Cornuejols et al. [13], with 100 facilities. We train and test on instances with 100 customers, and we evaluate on instances with 100 (Easy), 200 (Medium) and 400 (Hard) customers. These three benchmarks were chosen because they are challenging for state-of-the-art solvers, but also representative of the types of integer programming problems encountered in practice. In particular, set covering problems capture the quintessence of integer linear programming, since column generation formulations can be written for virtually any difficult discrete optimization problem. Throughout all experiments we use SCIP 6.0.1 as the backend solver, with a time limit

Table 2: Policy evaluation on separate instances in terms of solving time, number of wins (fastest method) over number of solved instances, and number of resulting B&B nodes (lower is better). For each problem, the models are trained on easy instances only. See Section 5.1 for definitions.

Model	Easy			Medium			Hard		
	Time	Wins	Nodes	Time	Wins	Nodes	Time	Wins	Nodes
FSB	20.19 ± 7.2%	0 / 100	16 ± 16.3%	282.14 ± 5.5%	0 / 100	215 ± 6.8%	3600.00 ± 0.0%	0 / 0	± %
RPB	13.38 ± 6.8%	1 / 100	<b>63</b> ± 24.0%	66.58 ± 6.1%	9 / 100	2327 ± 9.9%	1699.96 ± 4.1%	27 / 65	51 022 ± 7.0%
TREES	14.62 ± 7.4%	0 / 100	199 ± 11.7%	106.95 ± 8.0%	0 / 100	3043 ± 9.1%	2726.56 ± 2.1%	0 / 36	58 608 ± 5.3%
SVMRANK	13.33 ± 7.0%	1 / 100	157 ± 8.2%	89.63 ± 5.6%	0 / 100	2516 ± 4.0%	2401.43 ± 2.4%	0 / 48	42 824 ± 2.9%
LMART	<b>12.20</b> ± 7.3%	<b>59</b> / 100	161 ± 8.4%	72.07 ± 6.9%	12 / 100	2584 ± 5.4%	2177.72 ± 3.5%	0 / 54	48 032 ± 6.2%
GCNN	12.25 ± 7.6%	39 / 100	130 ± 7.5%	<b>59.40</b> ± 8.6%	<b>79</b> / 100	<b>1845</b> ± 3.4%	<b>1680.59</b> ± 5.1%	<b>40</b> / 64	<b>34 527</b> ± 5.2%
Set Covering									
FSB	7.27 ± 13.6%	0 / 100	5 ± 20.6%	92.49 ± 12.5%	0 / 100	72 ± 19.9%	1845.19 ± 5.2%	0 / 67	395 ± 8.0%
RPB	4.49 ± 13.6%	3 / 100	<b>8</b> ± 26.0%	18.45 ± 6.6%	0 / 100	<b>630</b> ± 25.5%	140.13 ± 7.8%	13 / 100	5440 ± 12.5%
TREES	3.58 ± 12.8%	0 / 100	82 ± 16.8%	23.67 ± 10.0%	0 / 100	944 ± 12.4%	481.11 ± 14.3%	0 / 95	10 752 ± 16.7%
SVMRANK	3.58 ± 14.4%	0 / 100	71 ± 16.3%	25.81 ± 11.1%	0 / 100	864 ± 15.0%	401.08 ± 8.7%	0 / 98	6353 ± 10.3%
LMART	<b>2.86</b> ± 13.2%	66 / 100	70 ± 16.3%	15.23 ± 10.1%	3 / 100	849 ± 13.3%	227.44 ± 8.7%	1 / 100	6878 ± 9.9%
GCNN	2.88 ± 12.6%	31 / 100	64 ± 13.6%	<b>11.23</b> ± 7.9%	<b>97</b> / 100	661 ± 11.4%	<b>118.74</b> ± 9.5%	<b>86</b> / 100	<b>4912</b> ± 11.8%
Combinatorial Auction									
FSB	30.86 ± 14.1%	5 / 100	8 ± 39.1%	237.14 ± 11.1%	3 / 97	66 ± 16.0%	1231.37 ± 11.2%	1 / 92	81 ± 11.5%
RPB	28.12 ± 12.4%	23 / 100	<b>13</b> ± 35.2%	182.31 ± 9.9%	1 / 100	<b>127</b> ± 25.4%	829.54 ± 10.4%	3 / 100	<b>149</b> ± 14.9%
TREES	28.88 ± 13.7%	15 / 100	105 ± 22.9%	191.95 ± 11.9%	0 / 100	481 ± 16.4%	895.37 ± 12.9%	5 / 100	495 ± 19.2%
SVMRANK	26.43 ± 12.8%	11 / 100	89 ± 22.5%	152.28 ± 8.1%	20 / 100	373 ± 14.4%	<b>726.79</b> ± 10.6%	25 / 100	395 ± 8.9%
LMART	26.21 ± 12.3%	13 / 100	88 ± 21.2%	149.60 ± 8.6%	23 / 100	367 ± 13.2%	733.48 ± 10.3%	31 / 100	395 ± 7.5%
GCNN	<b>26.01</b> ± 11.9%	<b>33</b> / 100	82 ± 20.8%	<b>147.22</b> ± 9.5%	<b>53</b> / 100	365 ± 16.8%	761.88 ± 11.7%	<b>35</b> / 100	388 ± 8.8%
Capacitated Facility Location									

Table 3: Ablation study of our GCNN model on the set covering problem. Sum convolutions generalize better to larger instances, especially when combined with a prenorm layer.

Model	Accuracies			Easy			Medium			Hard		
	acc@1	acc@5	acc@10	time	wins	nodes	time	wins	nodes	time	wins	nodes
MEAN	65.5 ± 0.1	92.3 ± 0.1	<b>98.3</b> ± 0.1	12.6 ± 9%	12 / 100	130 ± 8%	60.7 ± 9%	25 / 100	1885 ± 4%	1833.5 ± 8%	11 / 60	65 490 ± 13%
SUM	65.3 ± 0.1	92.3 ± 0.0	98.2 ± 0.1	12.5 ± 8%	30 / 100	130 ± 7%	60.0 ± 9%	32 / 100	1883 ± 4%	1802.5 ± 7%	10 / 64	63 284 ± 9%
GCNN	<b>65.6</b> ± 0.1	<b>92.5</b> ± 0.1	<b>98.3</b> ± 0.1	<b>12.3</b> ± 8%	<b>58</b> / 100	130 ± 8%	<b>59.4</b> ± 9%	<b>43</b> / 100	<b>1845</b> ± 3%	<b>1680.6</b> ± 5%	<b>46</b> / 64	<b>55 691</b> ± 7%

of 1 hour. Following Karzan et al. [28], Fischetti and Monaci [16], and Khalil et al. [29], we allow cutting plane generation at the root node only. All other SCIP parameters are kept to default so as to make comparisons as fair and reproducible as possible.

**Baselines** We compare against a human-designed state-of-the-art branching rule: reliability pseudocost (RPB), a variant of hybrid branching [1] which is used by default in SCIP. For completeness, we report as well the performance of full strong branching (FSB), our slow expert. We also compare against three machine learning branchers: the learning-to-score approach of Alvarez et al. [4] (TREES) based on an ExtraTrees [17] model, as well as the learning-to-rank approaches from Khalil et al. [29] (SVMRANK) and Hansknecht et al. [23] (LMART), based on an SVMrank [27] and a LambdaMART [12] model, respectively. The TREES model uses variable-wise features from our bipartite state, obtained by concatenating the variable’s node features with edge and constraint node features statistics over its neighborhood. The SVMRANK and LMART models both use the original features proposed by Khalil et al. [29], which we re-implemented within SCIP. More training details for each machine learning method can be found in the supplementary materials.

**Training** We train the models on each benchmark separately. Namely, for each benchmark, we generate 100,000 branching samples extracted from 10,000 randomly generated instances for training, 20,000 branching samples from 2,000 instances for validation, and same for test. We report in Table 1 the test accuracy of each machine learning model over five seeds, as the percentage of times the highest ranked decision of the model (acc@1), one of the five highest (acc@5) and one of the ten highest (acc@10) is a variable which is given the highest strong branching score.

**Evaluation** Evaluation is performed for each problem difficulty (Easy, Medium, Hard) on 20 new instances using five different seeds<sup>1</sup>, which amounts to a total of 100 solving attempts per method. We report standard metrics for MILP benchmarking<sup>2</sup>, that is: the 1-shifted geometric mean of the solving times in seconds, including running times of unsolved instances without extra penalization (Time); the hardware-independent final node counts of instances that are solved by all baselines (Nodes); and the number of times each branching policy results in the fastest solving time, over the number of instances solved (Win). Policy evaluation results are displayed in Table 2. Note that we also report the average per-instance standard deviation, so “ $64 \pm 13.6\%$  nodes” means it took on average 64 nodes to solve an instance, and when solving one of those instances the number of nodes varied by 13.6% on average.

## 5.2 Comparative experiment

In terms of prediction accuracy (Table 1), GCNN clearly outperforms the baseline competitors on all three problems, while SVMRANK and LMART are on par with each other and the performance of TREES is the lowest.

At solving time (Table 2), the accuracy of each method is clearly reflected in the number of nodes required to solve the instances. Interestingly, the best method in terms of nodes is not necessarily the best in terms of total solving time, which also takes into account the computational cost of each branching policy, i.e., the feature extraction and inference time. Our GCNN clearly dominates overall, except on easy instances where the LMART approach offers a slightly better running time / accuracy trade-off. The SVMRANK approach, despite being on par with LMART in terms of number of nodes, is also slower.

Our GCNN model generalizes well to instances of size larger than seen during training. GCNN even outperforms SCIP’s default branching rule RPB in terms of running time in every configuration, and also in terms of nodes on medium and hard instances for setcover problems, and on hard instances for combinatorial auctions. As expected, the FSB expert brancher is not competitive in terms of running time, despite producing very small search trees.

For the first time in the literature a machine-learning-based approach is compared with an essentially full-fledged MILP solver. For this reason, the results are particularly impressive, and indicate that GCNN is a very serious candidate to be implemented within a MILP solver, as an additional tool to speed up mixed-integer linear programming solvers. Also, they suggest that more could be gained from a tight integration within a complex software, like any MILP solver is.

## 5.3 Ablation study

We present an ablation study of our proposed GCNN model on the set covering problem by comparing three variants of the convolution operation in (5): mean rather than sum convolutions (MEAN), sum convolutions without our prenorm layer (SUM) and finally sum convolutions with prenorm layers, which is the model we use throughout our experiments (GCNN).

Results on test instances are reported in Table 3. The solving performance of both variants MEAN and SUM is very similar to that of our baseline GCNN on small instances. On large instances however, the variants perform significantly worse in terms of both solving time and number of nodes, especially on hard instances. This empirical evidence supports our hypothesis that sum-convolutions offer a better architectural prior than mean-convolution from the task of learning to branch, and that our prenorm layer helps for stabilizing training and improving generalization.

## 6 Conclusion

We formulated branch-and-bound, the standard exact method for solving mixed-integer linear programs, as a Markov decision process. In this context, we proposed and evaluated a novel approach for tackling the branching problem, by expressing the state of the branch-and-bound process as a

<sup>1</sup>In addition to ML models which are re-trained with a different seed, all major MILP solvers have a parameter, *seed*, that randomizes some tie-breaking rules, so as to be able to report aggregated results over the same instance.

<sup>2</sup>See e.g. <http://plato.asu.edu/bench.html>



bipartite graph, which reduces the need for feature engineering by naturally leveraging the variable-constraint structure of MILP problems, and allows for the encoding of branching policies as a graph convolutional neural network. We demonstrated on three NP-hard problems that, by adopting a simple imitation learning scheme, the policies learned by a GCNN were outperforming previously proposed machine learning approaches for branching, and could also outperform the default branching strategy of SCIP, a modern open-source solver. Most importantly, we demonstrated that the learned policies could generalize to instance sizes larger than seen during training. This is essential since collecting strong branching decisions, hence training, can be computationally prohibitive on large instances. Our work indicates that the GCNN model, especially using sum convolutions with the proposed prenorm layer, is a good architectural prior for the task of branching in MILP.

In future work, we would like to assess the viability of our approach on a broader set on combinatorial problems, and also experiment with reinforcement learning methods for improving over the policies learned by imitation. Also, we believe that there is plenty of room for hybrid approaches combining traditional methods and machine learning for branching, and we would like to dig deeper into the learned policies in order to extract some knowledge of interest for the MILP community.

## Acknowledgements

We would like to thank Ambros Gleixner and Benjamin Müller for enlightening discussions and technical help regarding SCIP, as well as Gonzalo Muñoz, Aleksandr Kazachkov and Giulia Zarpellon for insightful discussions on variable selection. We also thank Jason Jo, Meng Qu and Mike Pieper for their helpful comments on the structure of the paper.

## References

- [1] Tobias Achterberg and Timo Berthold. Hybrid branching. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2009.
- [2] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. *Facets of Combinatorial Optimization*, pages 449–481, 2013.
- [3] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2005.
- [4] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29:185–195, 2017.
- [5] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Finding cuts in the tsp. Technical report, DIMACS, 1995.
- [6] Karl J Åström. Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10:174–205, 1965.
- [7] Egon Balas and Andrew Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial Optimization*, pages 37–60. Springer, 1980.
- [8] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch. In *Proceedings of the International Conference on Machine Learning*, 2018.
- [9] Mislav Balunovic, Pavol Bielik, and Martin Vechev. Learning to solve SMT formulas. In *Advances in Neural Information Processing Systems 31*, pages 10338–10349, 2018.
- [10] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d’horizon. *arXiv preprint arXiv:1811.06128*, 2018.
- [11] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. In *Proceedings of the International Conference on Learning Representations*, 2014.

- [12] Christopher JC Burges. From RankNet to LambdaRank to LambdaMART: An Overview. Technical report, Microsoft Research, 2010.
- [13] G. Cornuejols, R. Sridharan, and J.M. Thizy. A comparison of heuristics and relaxations for the capacitated plant location problem. *European Journal of Operational Research*, 50(3):280 – 297, 1991. ISSN 0377-2217. doi: [https://doi.org/10.1016/0377-2217\(91\)90261-S](https://doi.org/10.1016/0377-2217(91)90261-S). URL <http://www.sciencedirect.com/science/article/pii/037722179190261S>.
- [14] Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash: Dynamic approach for switching heuristics. *European Journal of Operational Research*, 248:943–953, 2016.
- [15] David K. Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems* 28, pages 2224–2232, 2015.
- [16] Matteo Fischetti and Michele Monaci. Branching on nonchimerical fractionalities. *Operations Research Letters*, 40:159 – 164, 2012.
- [17] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63:3–42, 2006.
- [18] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the Thirty-Fourth International Conference on Machine Learning*, pages 1263–1272, 2017.
- [19] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1263–1272, 2017.
- [20] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 6.0. Zib-report, Zuse Institute Berlin, July 2018.
- [21] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, pages 249–256, 2010.
- [22] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, volume 2, pages 729–734, 2005.
- [23] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. arXiv:1805.01415, 2018.
- ✓ [24] He He, Hal III Daumé, and Jason Eisner. Learning to search in branch and bound algorithms. In *Advances in Neural Information Processing Systems* 27, pages 3293–3301, 2014.
- ✓ [25] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.
- [26] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys*, 50:21, 2017.
- [27] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 133–142. ACM, 2002.

- [28] Fatma K. Karzan, George L. Nemhauser, and Martin W. P. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1:249–293, 2009.
- [29] Elias B. Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. **Learning to branch in mixed integer programming**. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI’16, pages 724–731, 2016.
- [30] Elias B. Khalil, Bistra Dilkina, George L. Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 659–666, 2017.
- [31] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations*, 2017.
- [32] Alex Kulesza and Fernando Pereira. Structured learning with approximate inference. In *Advances in Neural Information Processing Systems 21*, pages 785–792, 2008.
- [33] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960.
- [34] Gil Lederman, Markus N Rabe, and Sanjit A Seshia. Learning heuristics for automated reasoning through deep reinforcement learning. arXiv:1807.08058, 2018.
- [35] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2Nd ACM Conference on Electronic Commerce*, EC ’00, pages 66–76, New York, NY, USA, 2000. ACM. ISBN 1-58113-272-7. doi: 10.1145/352871.352879. URL <http://doi.acm.org/10.1145/352871.352879>.
- [36] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In *Proceedings of the International Conference on Learning Representations*, 2016.
- [37] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140, 2016.
- [38] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1999.
- [39] Andrea Lodi and Giulia Zarpellon. **On learning and branching: a survey**. *TOP*, 25:207–236, 2017.
- [40] miplib2017. MIPLIB 2017, 2018. <http://miplib.zib.de>.
- [41] Vangelis Th. Paschos, editor. *Applications of Combinatorial Optimization*. Mathematics and Statistics. Wiley-ISTE, 2nd edition edition, 2014.
- [42] Jagat Patel and John W. Chinneck. Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Mathematical Programming*, 110:445–474, 2007.
- [43] Y. Pochet and L.A. Wolsey. *Production planning by mixed integer programming*. Springer Science and Business Media, 2006.
- [44] Dean A Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3:88–97, 1991.
- [45] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20:61–80, Jan 2009.
- [46] Gourab Kundu Srikumar, Vivek and Dan Roth. On amortizing inference cost for structured prediction. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1114–1124, 2012.

- [47] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT Press, second edition, 2018.
- [48] Jaakkola T. S. Wainwright, M. J. and A. S. Willsky. MAP estimation via agreement on trees: message-passing and linear programming. *IEEE transactions on information theory*, 51:3697–3717, 2005.
- [49] Laurence A. Wolsey. *Integer Programming*. Wiley-Blackwell, 1988.
- [50] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.