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
`n.ferroni@specialvideo.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 [42], but also in machine learning, such as in structured prediction and maximum a posteriori inference [51; 34; 49]. 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 [52]. 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) [35] is the exact method of choice. Branch-and-bound recursively partitions the solution space into a search tree, and computes relaxation

arXiv:1906.01629v3 [cs.LG] 30 Oct 2019

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 [41]. 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 [21]. 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 [44], 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 [41], 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 [30; 4; 24], 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 (GCNN), which allows us to exploit the natural bipartite graph representation of MILP problems, thereby reducing the amount of 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 [30; 24]. We evaluate our approach on four classes of NP-hard problems, namely set covering, combinatorial auction, capacitated facility location and maximum independent set. We compare against the previously proposed approaches of Khalil et al. [30], Alvarez et al. [4] and Hansknecht et al. [24], 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 that use 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. [30], 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. [24] who learn a branching rule 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 with a differently formulated learning problem. Namely, Khalil et al. [30] and Hansknecht et al. [24] treat it as a ranking problem and learn a 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 activated. We compare against these approaches in Section 5.

Other works have considered using graph convolutional neural networks in the context of approximate combinatorial optimization, where the objective is to find good solutions quickly, without seeking any optimality guarantees. The first work of this nature was by Khalil et al. [31], who proposed a GCNN model for learning greedy heuristics on several collections of combinatorial optimization

problems defined on graphs. This was followed by Selsam et al. [47], who proposed a recurrent GCNN model, NeuroSAT, which can be interpreted as an approximate SAT solver when trained to predict satisfiability. Such works provide additional evidence that GCNNs can effectively capture structural characteristics of combinatorial optimization problems.

Other works consider using machine learning to improve variable selection in branch-and-bound, without directly learning a branching policy. Di Liberto et al. [15] 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 candidate variables, and propose to learn a weighting of different existing scores to combine their strengths. Other works learn variable selection policies, but for algorithms less general than B&B. Liang et al. [39] learn a variable selection policy for SAT solvers using a bandit approach, and Lederman et al. [36] 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 mixed-integer linear programs. 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. [25] learn a node selection heuristic by imitation learning of the oracle procedure that expands the node whose feasible set contains the optimal solution, while Song et al. [48] learn node selection and pruning heuristics by imitation learning of shortest paths to good feasible solutions, and Khalil et al. [32] 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

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

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 $\mathbf{x}^\star$,

$$x_i \leq \lfloor x_i^\star \rfloor \lor x_i \geq \lceil x_i^\star \rceil, \quad \exists i \leq p \mid x_i^\star \notin \mathbb{Z}, \tag{2}$$

where $\lfloor . \rfloor$ and $\lceil . \rceil$ 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 $u_i = \lfloor x_i^\star \rfloor$ in the left child and $l_i = \lceil x_i^\star \rceil$ in the right child.

The branch-and-bound algorithm [52, 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
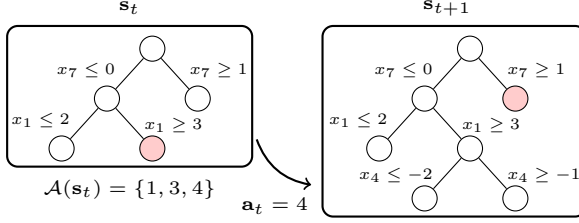
Figure 1: B&B variable selection as a Markov decision process. On the left, a state $\mathbf{s}_t$ comprised of the branch-and-bound tree, with a leaf node chosen by the solver to be expanded next (in pink). On the right, a new state $\mathbf{s}_{t+1}$ resulting from branching on the variable $\mathbf{a}_t = x_4$.

the core of modern combinatorial optimization solvers, and have been the focus of extensive research [40; 43; 1; 17]. 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 [43] or a hand-crafted combination of the two. For a 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. [25], the sequential decisions made during B&B can be assimilated to a Markov decision process [26]. Consider the solver to be the environment, and the brancher the agent. At the tth decision the solver is in a state $\mathbf{s}_t$, 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 $\mathbf{a}_t$ among all fractional variables $\mathcal{A}(\mathbf{s}_t) \subseteq \{1, \ldots, p\}$ at the currently focused node, according to a policy $\pi(\mathbf{a}_t \mid \mathbf{s}_t)$. 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 $\mathbf{s}_{t+1}$, 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 $\tau = (\mathbf{s}_0, \ldots, \mathbf{s}_T) \in \mathcal{T}$ then depends on both the branching policy π and the remaining components of the solver,

$$p_\pi(\tau) = p(\mathbf{s}_0) \prod_{t=0}^{T-1} \sum_{\mathbf{a} \in \mathcal{A}(\mathbf{s}_t)} \pi(\mathbf{a} \mid \mathbf{s}_t) p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}).$$

A natural approach to find good branching policies is reinforcement learning, with a carefully designed reward function. 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 [50]. 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 [27].
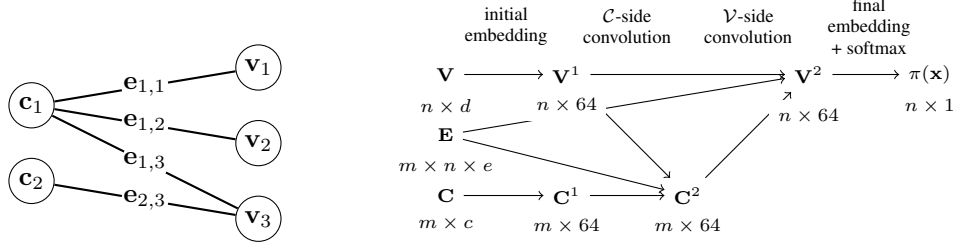
4

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

4.1 Imitation learning

We train by behavioral cloning [45] 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 $\mathcal{D} = \{(\mathbf{s}_i, \mathbf{a}_i^\star)\}_{i=1}^N$, and then learn our policy by minimizing the cross-entropy loss

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathbf{s}, \mathbf{a}^*) \in \mathcal{D}} \log \pi_\theta(\mathbf{a}^* \mid \mathbf{s}). \tag{3}$$

4.2 State encoding

We encode the state $\mathbf{s}_t$ 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 $\pi_\theta(\mathbf{a}|\mathbf{s}_t)$ as a graph convolutional neural network [23; 46; 12]. 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 [16; 19], program verification [38], and document classification in citation networks [33]. 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 $\mathbf{s}_t = (\mathcal{G}, \mathbf{C}, \mathbf{V}, \mathbf{E})$ 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 form

$$\mathbf{c}_i \leftarrow \mathbf{f}_{\mathcal{C}}\Big(\mathbf{c}_i, \sum_{j}^{(i,j)\in\mathcal{E}} \mathbf{g}_{\mathcal{C}}\left(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}\right)\Big), \qquad \mathbf{v}_j \leftarrow \mathbf{f}_{\mathcal{V}}\Big(\mathbf{v}_j, \sum_{i}^{(i,j)\in\mathcal{E}} \mathbf{g}_{\mathcal{V}}\left(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}\right)\Big) \qquad (4)$$

for all $i \in \mathcal{C}$, $j \in \mathcal{V}$, where $\mathbf{f}_{\mathcal{C}}$, $\mathbf{f}_{\mathcal{V}}$, $\mathbf{g}_{\mathcal{C}}$ and $\mathbf{g}_{\mathcal{V}}$ are 2-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 [33]. As noted by Xu et al. [53] 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 [22], 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 (4). The β and σ 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 four 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. [37, 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). Our third benchmark is comprised of capacitated facility location instances generated following Cornuejols et al. [14], 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. Finally, our fourth benchmark is comprised of maximum independent set instances on Erdős-Rényi random graphs, generated following the procedure of Bergman et al. [11, 4.6.4] with affinity set to 4. We train and test on instances with graphs of 500 nodes, and we evaluate on instances with 500 (Easy), 1000 (Medium) and 1500 nodes (Hard). These four 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 of 1 hour. Following Karzan et al. [29], Fischetti and Monaci [17] and Khalil et al. [30] we allow cutting plane generation at the root node only, and deactivate solver restarts. 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,

Table 1: Imitation learning accuracy on the test sets.

model	Set Covering			Combinatorial Auction			Capacitated Facility Location			Maximum Independent Set		
	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10
TREES	51.8±0.3	80.5±0.1	91.4±0.2	52.9±0.3	84.3±0.1	94.1±0.1	63.0±0.4	97.3±0.1	99.9±0.0	30.9±0.4	47.4±0.3	54.6±0.3
SVMRANK	57.6±0.2	84.7±0.1	94.0±0.1	57.2±0.2	86.9±0.2	95.4±0.1	67.8±0.1	98.1±0.1	99.9±0.0	48.0±0.6	69.3±0.2	78.1±0.2
LMART	57.4±0.2	84.5±0.1	93.8±0.1	57.3±0.3	86.9±0.1	95.3±0.1	68.0±0.2	98.0±0.0	99.9±0.0	48.9±0.3	68.9±0.4	77.0±0.5
GCNN	**65.5**±0.1	**92.4**±0.1	**98.2**±0.0	**61.6**±0.1	**91.0**±0.1	**97.8**±0.1	**71.2**±0.2	**98.6**±0.1	99.9±0.0	**56.5**±0.2	**80.8**±0.3	**89.0**±0.1

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	17.30 ± 6.1%	0/100	17 ±13.7%	411.34 ± 4.3%	0/90	171 ± 6.4%	3600.00 ± 0.0%	0/0	n/a ± n/a %
RPB	8.98 ± 4.8%	0/100	**54** ±20.8%	60.07 ± 3.7%	0/100	1741 ± 7.9%	1677.02 ± 3.0%	4/65	47 299 ± 4.9%
TREES	9.28 ± 4.9%	0/100	187 ± 9.4%	92.47 ± 5.9%	0/100	2187 ± 7.9%	2869.21 ± 3.2%	0/35	59 013 ± 9.3%
SVMRANK	8.10 ± 3.8%	1/100	165 ± 8.2%	73.58 ± 3.1%	0/100	1915 ± 3.8%	2389.92 ± 2.3%	0/47	42 120 ± 5.4%
LMART	7.19 ± 4.2%	14/100	167 ± 9.0%	59.98 ± 3.9%	0/100	1925 ± 4.9%	2165.96 ± 2.0%	0/54	45 319 ± 3.4%
GCNN	**6.59** ± 3.1%	**85**/100	134 ± 7.6%	**42.48** ± 2.7%	**100**/100	**1450** ± 3.3%	**1489.91** ± 3.3%	**66**/70	**29 981** ± 4.9%

Set Covering

Model	Easy			Medium			Hard		
FSB	4.11 ±12.1%	0/100	6 ±30.3%	86.90 ± 12.9%	0/100	72 ±19.4%	1813.33 ± 5.1%	0/68	400 ± 7.5%
RPB	2.74 ± 7.8%	0/100	**10** ±32.1%	17.41 ± 6.6%	0/100	689 ±21.2%	136.17 ± 7.9%	13/100	5511 ±11.7%
TREES	2.47 ± 7.3%	0/100	86 ±15.9%	23.70 ± 11.2%	0/100	976 ±14.4%	451.39 ±14.6%	0/95	10 290 ±16.2%
SVMRANK	2.31 ± 6.8%	0/100	77 ±15.0%	23.10 ± 9.8%	0/100	867 ±13.4%	364.48 ± 7.7%	0/98	6329 ± 7.7%
LMART	**1.79** ± 6.0%	**75**/100	77 ±14.9%	14.42 ± 9.5%	1/100	873 ±14.3%	222.54 ± 8.6%	0/100	7006 ± 6.9%
GCNN	1.85 ± 5.0%	25/100	70 ±12.0%	**10.29** ± 7.1%	**99**/100	**657** ±12.2%	**114.16** ±10.3%	**87**/100	**5169** ±14.9%

Combinatorial Auction

Model	Easy			Medium			Hard		
FSB	30.36 ±19.6%	4/100	14 ±34.5%	214.25 ± 15.2%	1/100	76 ±15.8%	742.91 ± 9.1%	15/90	55 ± 7.2%
RPB	26.55 ±16.2%	9/100	**22** ±31.9%	156.12 ± 11.5%	8/100	**142** ±20.6%	631.50 ± 8.1%	14/96	**110** ±15.5%
TREES	28.96 ±14.7%	3/100	135 ±20.0%	159.86 ± 15.3%	3/100	401 ±11.6%	671.01 ±11.1%	1/95	381 ±11.1%
SVMRANK	23.58 ±14.1%	11/100	117 ±20.5%	130.86 ± 13.6%	13/100	348 ±11.4%	586.13 ±10.0%	21/95	321 ± 8.8%
LMART	23.34 ±13.6%	16/100	117 ±20.7%	128.48 ± 15.4%	23/100	349 ±12.9%	582.38 ±10.5%	15/95	314 ± 7.0%
GCNN	**22.10** ±15.8%	**57**/100	107 ±21.4%	**120.94** ± 14.2%	**52**/100	339 ±11.8%	**563.36** ±10.7%	**30**/95	338 ±10.9%

Capacitated Facility Location

Model	Easy			Medium			Hard		
FSB	23.58 ±29.9%	9/100	7 ±35.9%	1503.55 ± 20.9%	0/74	38 ±28.2%	3600.00 ± 0.0%	0/0	n/a ± n/a %
RPB	8.77 ±11.8%	7/100	20 ±36.1%	**110.99** ± 24.4%	41/100	**729** ±37.3%	2045.61 ±18.3%	22/42	**2675** ±24.0%
TREES	10.75 ±22.1%	1/100	76 ±44.2%	1183.37 ± 34.2%	1/47	4664 ±45.8%	3565.12 ± 1.2%	0/3	38 296 ± 4.1%
SVMRANK	8.83 ±14.9%	2/100	46 ±32.2%	242.91 ± 29.3%	1/96	546 ±26.0%	2902.94 ± 9.6%	1/18	6256 ±15.1%
LMART	7.31 ±12.7%	30/100	52 ±38.1%	219.22 ± 36.0%	15/91	747 ±35.1%	3044.94 ± 7.0%	0/12	8893 ± 3.5%
GCNN	**6.43** ±11.6%	**51**/100	43 ±40.2%	192.91 ±110.2%	**42**/82	1841 ±88.0%	**2024.37** ±30.6%	**25**/29	2997 ±26.3%

Maximum Independent Set

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 [18] model, as well as the learning-to-rank approaches from Khalil et al. [30] (SVMRANK) and Hansknecht et al. [24] (LMART), based on an SVMrank [28] and a LambdaMART [13] 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. [30], 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 (see supplementary materials for details). 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[1], which amounts to a total of 100 solving attempts per method. We report standard metrics for MILP benchmarking[2], 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 four 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. The SVMRANK approach, despite being slightly better than LMART in terms of number of nodes, is also slower due to a worse running time / number of nodes trade-off. Our GCNN model clearly dominates overall, except on combinatorial auction (Easy) and maximum independent set (Medium) instances, where LMART and RPB are respectively faster.

Our GCNN model generalizes well to instances of size larger than seen during training, and outperforms SCIP's default branching rule RPB in terms of running time in almost every configuration. In particular and strikingly, it significantly outperforms RPB in terms of nodes on medium and hard instances for setcover and combinatorial auction problems. As expected, the FSB expert brancher is not competitive in terms of running time, despite producing very small search trees. The maximum independent set problem seems particularly challenging for generalization, as all machine learning approaches report a lower number of solved instances than the default RPB brancher, and GCNN, despite being the fastest machine learning approach overall, exhibits a high variability both in terms of time and number of nodes.

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 (4): 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.

---

[1]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.

[2]See e.g. http://plato.asu.edu/bench.html

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.

	Accuracies			Easy			Medium			Hard		
Model	acc@1	acc@5	acc@10	time	wins	nodes	time	wins	nodes	time	wins	nodes
MEAN	65.4 ±0.1	**92.4** ±0.1	**98.2** ±0.0	6.7 ±3%	13/100	134 ±6%	43.7 ±3%	19/100	1894 ±4%	1593.0 ±4%	6/70	62 227 ±6%
SUM	65.5 ±0.2	92.3 ±0.2	98.1 ±0.1	**6.6** ±3%	27/100	134 ±6%	**42.5** ±3%	**45**/100	1882 ±4%	1511.7 ±3%	22/70	57 864 ±4%
GCNN	**65.5** ±0.1	**92.4** ±0.1	**98.2** ±0.0	**6.6** ±3%	**60**/100	134 ±8%	**42.5** ±3%	36/100	**1870** ±3%	**1489.9** ±3%	**42**/70	**56 348** ±5%

6 Discussion

The objective of branch-and-bound is to solve combinatorial optimization problems as fast as possible. Branching policies must therefore balance the quality of decisions taken with the time spent to take each decision. An extreme example of this tradeoff is strong branching: this policy takes excellent decisions leading to low number of nodes overall, but every decision-making step is so slow that the overall running time is not competitive. Early experiments showed that we could take better decisions and decrease the number of nodes slightly on average by training a GCNN policy with more layers or with a larger embedding size. However, this would also lead to increased computational costs for inference and slightly larger times at each decision, and in the end increased solving times on average. The policy architecture we propose is thus a compromise between learning capacity and inference speed, something that is not traditionally a concern within the machine learning community.

Another concern among the combinatorial optimization community is the ability of policies trained on small instances to generalize to larger instances. We were able to show that machine learning methods, and the GCNN model in particular, can generalize to fairly larger instances. However, in general it is expected that the improvement in performance decreases as our model is evaluated on progressively larger problems, as can already be observed from Table 2. In early experiments with even larger instances (huge), we observed a performance drop for the model trained on our small instances. This could presumably be remedied by training on larger instances in the first place, and indeed a model trained on medium instances did perform well on those huge instances again. In any case, there are limits as to the generalization ability of any learned branching policy, and since the limit is likely very dependent on the problem structure, it is difficult to give any precise quantitative estimates a priori. This desirable ability to generalize outside of the training distribution, sometimes termed transfer learning, is also not a traditional concern in the machine learning community.

7 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 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 four 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

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 M. 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:1811.06128*, 2018.

[11] David Bergman, Andre A. Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision diagrams for optimization*. Springer, 2016.

[12] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *Proceedings of the Second International Conference on Learning Representations*, 2014.

[13] Christopher J. C. Burges. From RankNet to LambdaRank to LambdaMART: An Overview. Technical report, Microsoft Research, 2010.

[14] Gerard Cornuejols, Ramaswami Sridharan, and Jean Michel Thizy. A comparison of heuristics and relaxations for the capacitated plant location problem. *European Journal of Operational Research*, 50:280 – 297, 1991.

[15] 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.

[16] 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.

[17] Matteo Fischetti and Michele Monaci. Branching on nonchimerical fractionalities. *Operations Research Letters*, 40:159 – 164, 2012.

[18] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63:3–42, 2006.

[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 Thirty-Fourth 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] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Technical report, Optimization Online, August 2019.

[22] 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*, pages 249–256, 2010.

[23] Marco Gori, Gabriele Monfardini, and Franco 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.

[24] 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.

[25] 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.

[26] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[27] Ahmed Hussein, Mohamed M. Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys*, 50:21, 2017.

[28] 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, 2002.

[29] 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.

[30] 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*, pages 724–731, 2016.

[31] Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems 30*, pages 6348–6358, 2017.

[32] 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.

[33] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the Fifth International Conference on Learning Representations*, 2017.

[34] Alex Kulesza and Fernando Pereira. Structured learning with approximate inference. In *Advances in Neural Information Processing Systems 21*, pages 785–792, 2008.

[35] Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[36] Gil Lederman, Markus N. Rabe, and Sanjit A. Seshia. Learning heuristics for automated reasoning through deep reinforcement learning. arXiv:1807.08058, 2018.

[37] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the Second ACM Conference on Electronic Commerce*, pages 66–76, 2000.

[38] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *Proceedings of the Fourth International Conference on Learning Representations*, 2016.

[39] 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.

[40] Jeff T. Linderoth and Martin W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

[41] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *TOP*, 25:207–236, 2017.

[42] Vangelis T. Paschos, editor. *Applications of Combinatorial Optimization*. Mathematics and Statistics. Wiley-ISTE, second edition, 2014.

[43] Jagat Patel and John W. Chinneck. Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Mathematical Programming*, 110:445–474, 2007.

[44] Yves Pochet and Laurence A. Wolsey. *Production planning by mixed integer programming*. Springer Science and Business Media, 2006.

[45] Dean A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3:88–97, 1991.

[46] Franco Scarselli, Marco Gori, Ah C. Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009.

[47] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a sat solver from single-bit supervision. In *Proceedings of the Seventh International Conference on Learning Representations*, 2019.

[48] Jialin Song, Ravi Lanka, Albert Zhao, Yisong Yue, and Masahiro Ono. Learning to search via retrospective imitation. arXiv:1804.00846, 2018.

[49] Vivek G. K. Srikumar 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.

[50] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, second edition, 2018.

[51] Martin J. Wainwright, Tommi S. Jaakkola, and Alan S. Willsky. MAP estimation via agreement on trees: message-passing and linear programming. *IEEE Transactions on Information Theory*, 51:3697–3717, 2005.

[52] Laurence A. Wolsey. *Integer Programming*. Wiley-Blackwell, 1988.

[53] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *Proceedings of the Seventh International Conference on Learning Representations*, 2019.

Exact Combinatorial Optimization
with Graph Convolutional Neural Networks
Supplementary Materials

Maxime Gasse
Mila, Polytechnique Montréal
maxime.gasse@polymtl.ca

Didier Chételat
Polytechnique Montréal
didier.chetelat@polymtl.ca

Nicola Ferroni
University of Bologna
n.ferroni@specialvideo.it

Laurent Charlin
Mila, HEC Montréal
laurent.charlin@hec.ca

Andrea Lodi
Mila, Polytechnique Montréal
andrea.lodi@polymtl.ca

1 Dataset collection details

For each benchmark problem, namely, set covering, combinatorial auction, capacitated facility location and maximum independent set, we generate 10,000 random instances for training, 2,000 for validation, and 3x20 for testing (20 easy instances, 20 medium instances, and 20 hard instances). In order to obtain our datasets of state-action pairs $\{(\mathbf{s}_i, \mathbf{a}_i)\}$ for training and validation, we pick an instance from the corresponding set (training or validation), solve it with SCIP, each time with a new random seed, and record new node states and strong branching decision during the branch-and-bound process. We continue processing new instances by sampling with replacement, until the desired number of node samples is reached, that is, 100,000 samples for training, and 20,000 for validation. Note that this way the whole set of training and validation instances is not necessarily used to generate samples. We report the number of training instances actually used for each problem in Table 1.

Table 1: Number of training instances solved by SCIP for obtaining 100,000 training samples (state-action pairs). We report both the total number of SCIP solves and the number of unique instances solved, since instances are sampled with replacement.

	Set Covering	Combinatorial Auction	Capacitated Facility Location	Maximum Independent Set
total	7771	11 322	7159	6198
unique	5335	6661	5046	4349

Note that the strong branching rule, as implemented in the SCIP solver, does not only provides branching decisions, but also triggers side-effects which change the state of the solver itself. In order to use strong branching as an oracle only, when generating our training samples, we re-implemented a vanilla version of the full strong branching rule in SCIP, named *vanillafullstrong*. This version of full strong branching also facilitates the extraction of strong branching scores for training the ranking-based and regression-based machine learning competitors, and will be included by default in the next version of SCIP.

2 Training details

2.1 GCNN

As described in the main paper, for our GCNN model we record strong branching decisions ($\mathbf{a}$) and extract bipartite state representations ($\mathbf{s}$) during branch-and-bound on a collection of training instances. This yields a training dataset of state-action pairs $\{(\mathbf{s}_t, \mathbf{a}_t)\}$. We train our model with the Tensorflow [1] library, with the same procedure throughout all experiments. We first pretrain the prenorm layers as described in the main paper. We then minimize a cross-entropy loss using Adam [4] with minibatches of size 32 and an initial learning rate of 1e-3. We divide the learning rate by 5 when the validation loss does not improve for 10 epochs, and stop training if it does not improve for 20. All experiments are performed on a machine with an Intel Xeon Gold 6126 CPU at 2.60GHz and an Nvidia Tesla V100 GPU.

A list of the features included in our bipartite state representation is given as Table 2. For more details the reader is referred to the source code at `https://github.com/ds4dm/learn2branch`.

Table 2: Description of the constraint, edge and variable features in our bipartite state representation $\mathbf{s}_t = (\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$.

Tensor	Feature	Description
C	obj_cos_sim	Cosine similarity with objective.
	bias	Bias value, normalized with constraint coefficients.
	is_tight	Tightness indicator in LP solution.
	dualsol_val	Dual solution value, normalized.
	age	LP age, normalized with total number of LPs.
E	coef	Constraint coefficient, normalized per constraint.
V	type	Type (binary, integer, impl. integer, continuous) as a one-hot encoding.
	coef	Objective coefficient, normalized.
	has_lb	Lower bound indicator.
	has_ub	Upper bound indicator.
	sol_is_at_lb	Solution value equals lower bound.
	sol_is_at_ub	Solution value equals upper bound.
	sol_frac	Solution value fractionality.
	basis_status	Simplex basis status (lower, basic, upper, zero) as a one-hot encoding.
	reduced_cost	Reduced cost, normalized.
	age	LP age, normalized.
	sol_val	Solution value.
	inc_val	Value in incumbent.
	avg_inc_val	Average value in incumbents.

2.2 SVMrank and LambdaMART

For SVMrank and LambdaMART, at each node t we record strong branching ranks ρ_i for each candidate variable i, and extract the same variable-wise features ϕ_i^{Khalil} as Khalil et al. [3], with two modifications. First, because of differences between CPLEX and SCIP, it is difficult to reimplement single/double infeasibility statistics, but SCIP keeps track of left/right infeasibility statistics, namely the number of times branching on the variable led to a left (resp. right) infeasible child. Because these statistics capture similar aspects of the branch-and-bound tree, we swapped one for the other. Second, since in our setting at test time there is no separate data collection stage, statistics are computed on

2

past branching decisions at each time step t. Otherwise we follow [3], which suggest a zero-one feature normalization over the candidate, a.k.a. *query-based normalization*, and a binarization of the ranking labels around the 80th centile. This yields a training dataset of variable-wise state-rank pairs $\{(\phi_i^{\text{Khalil}}(\mathbf{s}_t), \rho_{i,t})\}$.

The SVMrank model is trained by minimizing a cost sensitive pairwise loss with a 2nd-order polynomial feature augmentation, and regularization on the validation set among $C \in \{0.001, 0.01, 0.1, 1\}$, as in the original implementation. The LambdaMART model uses 500 estimators, and is trained by maximizing a normalized discounted cumulative gain with default parameters, with early stopping using the validation set.

Note that in their SVMrank implementation, Khalil et al. [3] limit at each node the candidate variables to the ten with highest pseudocost, both for training and testing. This makes sense in their online setting, as the model is trained and applied after observing the behavior of strong branching on a few initial nodes, which initializes the pseudocosts properly. In our context, however, there is no initial strong branching phase, and as is to be expected we observed that applying this pseudocost filtering scheme degraded performance. Consequently, we do not filter, and rather train and test using the whole set of candidate variables at every node.

In addition, both the SVMrank and LambdaMART approaches suffer from poorer training scalability than stochastic gradient descent-based deep neural networks, and we found that training on our entire dataset was prohibitive. Indeed, training would have taken more than 500GB of RAM for SVMrank and more than 1 week for LambdaMART for even the smallest problem class. Consequently, we had to limit the size of the dataset, and chose to reduce the training set to 250,000 candidate variables, and the validation set to 100,000.

2.3 ExtraTrees

For ExtraTrees, we record strong branching scores σ_i for each candidate variable i, and extract variable-wise features ϕ_i^{simple} from our bipartite state $\mathbf{s}$ as follows. For every variable we keep the original variable node features $\mathbf{v}_i$, which we concatenate with the component-wise minimum, mean and maximum of the edge and constraint node features $(\mathbf{e}_{i,j}, \mathbf{c}_j)$ over its neighborhood. As a result, we obtain a training dataset of variable-wise state-score pairs $\{(\phi_i^{\text{simple}}(\mathbf{s}_t), \sigma_{i,t})\}$. The model is trained by mean-squared error minimization, while at test time branching is made on the variable with the highest predicted score, $i^* = \arg\max_i \hat{\sigma}(\phi_i(\mathbf{s}_t))$. As Alvarez et al. [2, p. 14] mention, with this model the inference time increases with the training set size. Additionally, even though in practice the ExtraTrees model training scales to larger datasets than the SVMrank and LambdaMART models, here also we ran into memory issues when training on our entire dataset. Consequently, for ExtraTrees we also chose to limit the training dataset to 250,000 candidate variables, a figure roughly in line with Alvarez et al. [2]. Note that we did not use the expert features proposed by Khalil et al. [3] for ExtraTrees, as those were resulting in degraded performance.

3   When are decisions hard for the GCNN policy?

To understand what kinds of decisions are taken by the trained GCNN policy beyond the imitation learning accuracy, it is insightful to look at when it is confident. In particular, we can take the entropy of our policy as a measure of confidence, and compute an analogous quantity from the strong branching scores, $\mathbb{H}(\text{strong branching}) = \log m$ for m the number of candidates with maximal strong branching score. Figure 1 shows that entropies at every decision point on the easy instances for the four problems roughly correlate. This suggests that the decisions where the GCNN hesitates are those that are intrinsically difficult, since on those the expert hesitates more as well.

4   Training set sizes for the machine learning methods

As discussed, computational limitations made training of machine learning competitors impossible on the full dataset that we used to train our GCNN policy. It is a limitation of those methods that they cannot benefit from as much training data as the proposed deep policy with the same computational budget, and conversely, the scalability of our proposed method to large training sets is a major advantage. Since there is little reason for artificially restricting any model to smaller datasets than
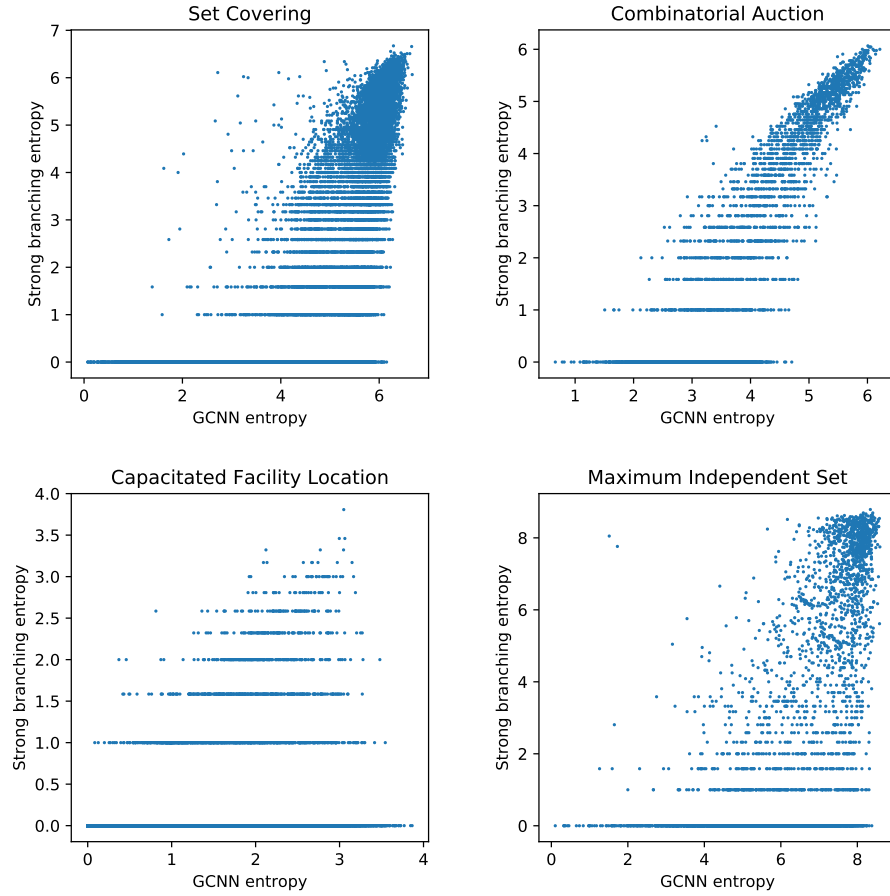
Figure 1: Entropy of the GCNN policy against the "entropy" of strong branching.

Table 3: Number of branching nodes (state-action pairs) used for training the machine learning competitors, that is, for obtaining 250,000 training and 100,000 validation samples (variable-score pairs). In comparison, the number of branching nodes in the complete dataset used to train the GCNN model is 100,000 for training, and 20,000 for validation.

	Set Covering	Combinatorial Auction	Capacitated Facility Location	Maximum Independent Set
training	1979	1738	6201	534
validation	782	696	2465	213

they can handle, especially since the state-of-the-art branching rule (reliability pseudocost) is not even a machine learning method, we did not do so in the paper.

Nonetheless, a natural question is whether the improvements in performance provided by the GCNN came solely from increased dataset size. To answer this question, we re-trained a GCNN model using the same amount of data as for the competitors, that is, 250,000 candidate variables for training, and 100,000 for validation. The size of the resulting datasets, from the point of view of the number of branching nodes recorded, is reported in Table 3. As can be seen in Table 4, the resulting model (GCNN-SMALL) still clearly outperforms the competitors in terms of accuracy. Thus although using more training samples improves performance, the improvements cannot be explained only by the amount of training data.

Table 4: Imitation learning accuracy on the test sets.

model	Set Covering			Combinatorial Auction			Capacitated Facility Location			Maximum Independent Set		
	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10
TREES	51.8±0.3	80.5±0.1	91.4±0.2	52.9±0.3	84.3±0.1	94.1±0.1	63.0±0.4	97.3±0.1	99.9±0.0	30.9±0.4	47.4±0.3	54.6±0.3
SVMRANK	57.6±0.2	84.7±0.1	94.0±0.1	57.2±0.2	86.9±0.2	95.4±0.1	67.8±0.1	98.1±0.1	99.9±0.0	48.0±0.6	69.3±0.2	78.1±0.2
LMART	57.4±0.2	84.5±0.1	93.8±0.1	**57.3**±0.3	86.9±0.2	95.3±0.1	68.0±0.2	98.0±0.0	99.9±0.0	48.9±0.3	68.9±0.4	77.0±0.5
GCNN-SMALL	**57.9**±1.0	**87.1**±0.6	**95.5**±0.3	55.0±1.6	**88.0**±0.6	**96.2**±0.1	**69.1**±0.1	**98.2**±0.0	99.9±0.0	**50.1**±1.2	**73.4**±0.6	**82.8**±0.6
GCNN	65.5±0.1	92.4±0.1	98.2±0.0	61.6±0.1	91.0±0.1	97.8±0.1	71.2±0.2	98.6±0.1	99.9±0.0	56.5±0.2	80.8±0.3	89.0±0.1

Table 5: Training time for each machine learning method, in hours.

	Set Covering	Combinatorial Auction	Capacitated Facility Location	Maximum Independent Set
TREES	0.05 ± 0.00	0.03 ± 0.00	0.16 ± 0.01	0.04 ± 0.00
SVMRANK	1.21 ± 0.01	1.17 ± 0.06	1.04 ± 0.03	1.19 ± 0.02
LMART	2.87 ± 0.23	2.47 ± 0.26	1.38 ± 0.15	2.16 ± 0.53
GCNN	14.45 ± 1.56	3.84 ± 0.33	18.18 ± 2.98	4.73 ± 0.85

5 Training and inference times

For completeness, we report in Table 5 the training time of each machine learning method, and in Table 6 their inference time per node. As can be observed, the GCNN model relies on less complicated features than the other methods, and therefore requires less computational time during feature extraction, at the cost of a higher prediction time.

Table 6: Inference time per node for each machine learning model, in milliseconds. We report both the time required to extract and compute features from SCIP (feat. extract), and the total inference time (total) which includes feature extraction and model prediction.

model	Easy		Medium		Hard	
	total	feat. extract	total	feat. extract	total	feat. extract
TREES	23.8 ± 4.0	11.2 ± 2.0	28.4 ± 4.5	15.7 ± 3.2	54.9 ±14.2	42.7 ±13.8
SVMRANK	16.6 ± 7.1	6.4 ± 4.3	23.1 ±11.1	9.4 ± 5.1	150.4 ±19.3	55.6 ±12.6
LMART	7.0 ± 3.5	6.0 ± 3.3	10.1 ± 4.8	8.8 ± 4.6	55.7 ±12.9	51.9 ±12.6
GCNN	5.5 ±13.1	1.1 ± 3.5	5.4 ± 4.6	1.5 ± 3.0	10.2 ±12.7	4.9 ±10.2

Set Covering

model	Easy		Medium		Hard	
TREES	16.1 ± 5.1	5.9 ± 1.8	25.3 ± 6.1	10.6 ± 4.7	30.2 ± 9.6	15.6 ± 7.8
SVMRANK	14.7 ±10.4	3.9 ± 4.8	31.6 ±20.2	8.3 ± 5.7	55.4 ±39.2	13.5 ± 9.4
LMART	4.5 ± 2.6	3.6 ± 2.3	9.6 ± 5.7	8.2 ± 5.4	15.4 ± 9.1	13.5 ± 8.6
GCNN	7.2 ±24.3	1.2 ± 5.7	4.4 ± 6.1	1.5 ± 4.2	5.1 ± 6.3	2.1 ± 5.3

Combinatorial Auction

model	Easy		Medium		Hard	
TREES	53.7 ± 2.0	49.1 ± 1.1	99.0 ± 6.2	93.4 ± 4.6	205.3 ± 4.9	199.7 ± 4.3
SVMRANK	13.3 ± 6.5	9.3 ± 5.8	21.8 ±14.5	17.9 ±14.0	80.4 ±70.9	74.1 ±70.5
LMART	9.7 ± 6.2	9.3 ± 6.1	18.4 ±14.1	17.9 ±13.9	83.5 ±80.4	82.9 ±80.3
GCNN	9.2 ±15.3	3.9 ± 3.3	14.8 ±18.7	7.5 ± 1.1	43.0 ±53.7	19.9 ±13.9

Capacitated Facility Location

model	Easy		Medium		Hard	
TREES	41.3 ± 7.8	12.0 ± 5.8	56.9 ±12.4	25.0 ±11.1	83.8 ±17.5	45.4 ±15.5
SVMRANK	45.5 ± 5.6	6.9 ± 5.4	96.5 ±12.8	19.4 ±11.8	200.2 ±17.8	35.4 ±17.5
LMART	8.3 ± 5.0	6.6 ± 5.0	22.1 ±11.2	19.2 ±11.2	38.8 ±16.0	34.8 ±15.9
GCNN	5.1 ± 7.4	2.3 ± 5.1	7.7 ±11.1	4.8 ±10.0	12.0 ±17.5	8.1 ±14.8

Maximum Independent Set

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath

Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Alejandro M. Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29:185–195, 2017.

[3] 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*, pages 724–731, 2016.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the Third International Conference on Learning Representations*, 2015.