

Parameterizing Branch-and-Bound Search Trees to Learn Branching Policies

Giulia Zarpellon,^{1*} Jason Jo,^{2,3} Andrea Lodi,¹ Yoshua Bengio^{2,3}

¹Polytechnique Montréal

²Mila

³Université de Montréal

giulia.zarpellon@polymtl.ca, jason.jo.research@gmail.com, andrea.lodi@polymtl.ca, yoshua.bengio@mila.quebec

Abstract

Branch and Bound (B&B) is the exact tree search method typically used to solve Mixed-Integer Linear Programming problems (MILPs). Learning branching policies for MILP has become an active research area, with most works proposing to imitate the strong branching rule and specialize it to distinct classes of problems. We aim instead at learning a policy that generalizes across heterogeneous MILPs: our main hypothesis is that parameterizing the state of the B&B search tree can aid this type of generalization. We propose a novel imitation learning framework, and introduce new input features and architectures to represent branching. Experiments on MILP benchmark instances clearly show the advantages of incorporating an explicit parameterization of the state of the search tree to modulate the branching decisions, in terms of both higher accuracy and smaller B&B trees. The resulting policies significantly outperform the current state-of-the-art method for “learning to branch” by effectively allowing generalization to generic unseen instances.

1 Introduction

Many problems arising from transportation, healthcare, energy and logistics can be formulated as Mixed-Integer Linear Programming (MILP) problems, i.e., optimization problems in which some decision variables represent discrete or indivisible choices. A MILP is written as

$$\min_x \{c^T x : Ax \geq b, x \geq 0, x_i \in \mathbb{Z} \forall i \in \mathcal{I}\}, \quad (1)$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c, x \in \mathbb{R}^n$ and $\mathcal{I} \subseteq \{1, \dots, n\}$ is the set of indices of variables that are required to be integral, while the other ones can be real-valued. Note that one can consider a MILP as defined by (c, A, b, \mathcal{I}) ; we do not assume any special combinatorial structure on the parameters c, A, b . While MILPs are in general \mathcal{NP} -hard, MILP solvers underwent dramatic improvements over the last decades (Lodi 2009; Achterberg and Wunderling 2013) and now achieve high-performance on a wide range of problems. The fundamental component of any modern MILP solver is Branch and Bound (B&B) (Land and Doig 1960), an *exact* tree search method. Following a divide-and-conquer approach, B&B partitions the search space by branching on

variables’ values and smartly uses bounds from problem relaxations to prune unpromising regions from the tree. The B&B algorithm actually relies on expertly-crafted *heuristic* rules for its two most fundamental decisions: *branching variable selection* (BVS) and *node selection*. In particular, BVS is a crucial factor for B&B’s success (Achterberg and Wunderling 2013), and will be the main focus of the present article.

Understanding why B&B works has been called “one of the mysteries of computational complexity theory” (Lipton and Regan 2012), and there currently is no mathematical theory of branching; to the best of our knowledge, the only attempt in formalizing BVS is the recent work of Le Bodic and Nemhauser. One central reason why B&B is difficult to formalize resides in its inherent exponential nature: millions of BVS decisions could be needed to solve a MILP, and a single bad one could result in a doubled tree size and no improvement in the search. Such a complex and data-rich setting, paired with a lack of formal understanding, makes B&B an appealing ground for machine learning (ML) techniques, which have lately been thriving in discrete optimization (Bengio, Lodi, and Prouvost 2018). In particular, there has been substantial effort towards “learning to branch”, i.e., in using ML methods to learn BVS policies (Lodi and Zarpellon 2017). Up to now, most works in this area focused on learning branching policies by supervision or imitation of *strong branching* (SB), a valid but expensive heuristic scheme (see Sections 2 and 5). The latest and state-of-the-art contribution to “learning to branch” (Gasse et al. 2019) frames BVS as a classification problem on SB expert decisions, and employs a graph-convolutional neural network (GCNN) to represent MILPs via their variable-constraint structure. The resulting branching policies improve on the solver by specializing SB to different classes of synthetic problems, and the attained generalization ability is to similar MILP instances (within the same class), possibly larger in formulation size.

The present work seeks a different type of generalization for a branching policy, namely a systematic generalization across *heterogeneous* MILPs, i.e., across problems not belonging to the same combinatorial class, without any restriction on the formulation’s structure and size. To achieve this goal, *we parameterize BVS in terms of B&B search trees*. On the one hand, information about the state of the B&B

*Corresponding author. Code, data and supplementary materials can be found at: <https://github.com/ds4dm/branch-search-trees>
Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

tree – abundant yet mostly unexploited by MILP solvers – was already shown to be useful to learn resolution patterns shared across general MILPs (Fischetti, Lodi, and Zarpellon 2019). On the other hand, the state of the search tree ought to have a central role in BVS – which ultimately decides how the tree is expanded and hence how the search itself proceeds. In practice, B&B continually interacts with other algorithmic components of the solver to effectively search the decision tree, and some algorithmic decisions may be triggered depending on which phase the optimization is in (Berthold, Hendel, and Koch 2017). In a highly integrated framework, a branching variable should thus be selected among the candidates based on its role in the search and its various components. Indeed, state-of-the-art heuristic branching schemes employ properties of the tree to make BVS decisions, and the B&B method equipped with such branching rules has proven to be successful across widely heterogeneous instances.

Motivated by these considerations, our main hypothesis is that MILPs share a higher order structure in the space of B&B search trees, and parameterized BVS policies should learn in this representational space. We setup a novel learning framework to investigate this idea. First of all, there is no natural input representation of this underlying space. Our first contribution is to craft input features of the variables that are candidates for branching: we aim at representing their roles in the search and its dynamic evolution. The dimensionality of such descriptions naturally changes with the number of candidates at every BVS step. The deep neural network (DNN) architecture that we propose learns a baseline branching policy (NoTree) from the candidate variables’ representations and effectively deals with varying input dimensions. Taking this idea further, we suggest that an explicit representation of the *state of the search tree* should condition the branching criteria, in order for it to flexibly adapt to the tree evolution. We contribute such tree-state parameterization, and incorporate it to the baseline architecture to provide context over the candidate variables at each given branching step. In the resulting policy (TreeGate) the tree state acts as a control mechanism to drive a top-down modulation (specifically, feature gating) of the highly mutable space of candidate variables representations. By training in our hand-crafted input space, the signal-to-noise ratio of the high-level branching structure shared amongst general MILPs is effectively increased, enabling our TreeGate policy to rapidly infer these latent factors and dynamically compose them via top-down modulation. In this sense, *we learn branching from parameterizations of B&B search trees that are shared among general MILPs*. To the best of our knowledge, the present work is the first attempt in the “learning to branch” literature to represent B&B search trees for branching, and to establish such a broad generalization paradigm covering many classes of MILPs.

We perform imitation learning (IL) experiments on a curated dataset of heterogeneous instances from standard MILP benchmarks. We employ as expert rule `relpscost`, the default branching scheme of the optimization solver SCIP (Gleixner et al. 2018), to which our framework is integrated. Machine learning experimental results clearly show

the advantage of the policy employing the tree state (TreeGate) over the baseline one (NoTree), the former achieving a 19% improvement in test accuracy. When plugged in the solver, both learned policies compare well with state-of-the-art branching rules. The evaluation of the trained policies in the solver also supports our idea that representing B&B search trees enables learning to branch across generic MILP instances: over test instances, the best TreeGate policy explores on average trees with 27% less nodes than the best NoTree one. In contrast, the GCNN framework of Gasse et al. that we use as benchmark does not appear to be able to attain such broad generalization goal: often the GCNN models fail to solve heterogeneous test instances, exploring search trees that are considerably bigger than those we obtain. The comparison thus remarks the advantage of our fundamentally new architectural paradigm – of representing candidates’ role in the search and using a tree-context to modulate BVS – which without training in a class-specific manner nor focusing on constraints structure effectively allows learning across generic MILPs.

2 Background

Simply put, the B&B algorithm iteratively partitions the solution space of a MILP (1) into sub-problems, which are mapped to nodes of a binary decision tree. At each node, integrality requirements for variables in \mathcal{I} are dropped, and a linear programming (LP) (continuous) relaxation of the problem is solved to provide a valid lower bound to the optimal value of (1). When the solution x^* of a node LP relaxation violates the integrality of some variables in \mathcal{I} , that node is further partitioned into two children by *branching on a fractional variable*. Formally, $\mathcal{C} = \{i \in \mathcal{I} : x_i^* \notin \mathbb{Z}\}$ defines the index set of *candidate variables* for branching at that node. The BVS problem consists in selecting a variable $j \in \mathcal{C}$ in order to *branch* on it, i.e., create child nodes according to the split

$$x_j \leq \lfloor x_j^* \rfloor \vee x_j \geq \lceil x_j^* \rceil. \quad (2)$$

Child nodes inherit a lower bound estimate from their parent, while (2) ensures x^* is removed from their solution spaces. After extending the tree, the algorithm moves on to select a new *open node*, i.e., a leaf yet to be explored (node selection): a new relaxation is solved, and new branchings happen. When x^* satisfies integrality requirements, then it is actually feasible for (1), and its value provides a valid upper bound to the optimal one. Maintaining global upper and lower bounds allows one to prune large portions of the search space. During the search, *final leaf nodes* are created in three possible ways: by integrality, when the relaxed solution is feasible for (1); by infeasibility of the sub-problem; by bounds, when the comparison of the node’s lower bound to the global upper one proves that its sub-tree is not worth exploring. An optimality certificate is reached when the global bounds converge. See (Wolsey 1998; Lodi 2009) for details on B&B and its combination with other components of a MILP solver.

Branching rules Usually, candidates are evaluated with respect to some scoring function, and j is chosen for branch-

ing as the (or a) score-maximizing variable (Achterberg, Koch, and Martin 2005). The most used criterion in BVS measures variables depending on the improvement of the lower bound in their (prospective) child nodes. The *strong branching* (SB) rule (Applegate et al. 1995) explicitly computes bound gains for \mathcal{C} . The procedure is expensive, but experimentally realizes trees with the least number of nodes. Instead, *pseudo-cost* (PC) (Benichou et al. 1971) maintains a history of variables’ branchings, averaging past improvements to get a proxy for the expected gain. Fast in evaluation, PC can behave badly due to uninitialization, so combinations of SB with PC have been developed. In *reliability branching*, SB is performed until PC scores for a variable are deemed reliable proxies of bound improvements. In *hybrid branching* (Achterberg and Berthold 2009), PC scores are combined with other ones measuring the variables’ role on inference and conflict clauses. Many other scoring criteria have been proposed, and some of them are surveyed in Lodi and Zarpellon from a ML perspective.

State-of-the-art branching rules can in fact be interpreted as mechanisms to score variables based on their effectiveness in different search components. While hybrid branching explicitly combines five scores reflecting variables’ behaviors in different search tasks, the evaluation performed by SB and PC can also be seen as a measure of how effective a variable is – in the single task of improving the bound from one parent node to its children. Besides, one can assume that the importance of different search functionalities should change dynamically during the tree exploration.¹ In this sense, our approach aims at learning a branching rule that takes into account variables’ roles in the search and the tree evolution itself to perform a more flexible BVS, adapted to the search stages.

3 Parameterizing B&B Search Trees

The central idea of our framework is to learn BVS by means of parameterizing the underlying space of B&B search trees. We believe this space can represent the complexity and the dynamism of branching in a way that is shared across heterogeneous problems. However, there are no natural parameterization of BVS or B&B search trees. To this end, our contribution is two-fold: 1) we propose hand-crafted input features to describe candidate variables in terms of their roles in the B&B process, and explicitly encode a “tree state” to provide a richer context to variable selection; 2) we design novel DNN architectures to integrate these inputs and learn BVS policies.

3.1 Hand-crafted Input Features

At each branching step t , we represent the set of variables that are candidates for branching by an input matrix $C_t \in \mathbb{R}^{25 \times |C_t|}$. To capture the multiple roles of a variable throughout the search, we describe each candidate $x_j, j \in C_t$ in terms of its bounds and solution value in the current sub-problem. We also feature statistics of a variable’s participation in various search components and in past branchings.

¹Indeed, a “dynamic factor” adjusts weights in the default branching scheme of SCIP (relpscost).

In particular, the scores that are used in the SCIP default hybrid-branching formula are part of C_t .

Additionally, we create a separate parameterization $Tree_t \in \mathbb{R}^{61}$ to describe the state of the search tree. We record information of the current node in terms of depth and bound quality. We also consider the growth rate and the composition of the tree, the evolution of global bounds, aggregated variables’ scores, statistics on feasible solutions and on bound estimates and depths of open nodes.

All features are designed to capture the dynamics of the B&B process linked to BVS decisions, and are efficiently gathered through a customized version of PySCIPOpt (Mather et al. 2016). Note that $\{C_t, Tree_t\}$ are defined in a way that is not explicitly dependent on the parameters of each instance (c, A, b, \mathcal{I}) . Even though C_t naturally changes its dimensionality at each BVS step t depending on the highly variable C_t , the fixed lengths of the vectors enable training among branching sets of different sizes (see 3.2). The representations evolve with the search: t-SNE plots (van der Maaten and Hinton 2008) in Figures 1(a) and 1(b) synthesize the evolution of $Tree_t$ throughout the B&B search, for two different MILP instances. The pictures clearly show the high heterogeneity of the branching data across different search stages. A detailed description of the hand-crafted input features is reported in the supplementary material (SM).

3.2 Architectures to Model Branching

We use parameterizations C_t as inputs for a baseline DNN architecture (NoTree). Referring to Figure 2, the 25-feature input of a candidate variable is first embedded into a representation with hidden size h ; subsequently, multiple layers reduce the dimensionality from h to an infimum INF by halving it at each step. The vector of length INF is then compressed by global average pooling into a single scalar. The $|C_t|$ dimension of C_t is conceived (and implemented) as a “batch dimension”: this makes it possible to handle branching sets of varying sizes, still allowing the parameters of the nets to be shared across problems. Ultimately, a *softmax* layer yields a probability distribution over the candidate set C_t , according to which a variable is selected for branching.

We incorporate the tree-state input to the baseline architecture to provide a search-based context over the mutable branching sets. Practically, $Tree_t$ is embedded in a series of subsequent layers with hidden size h . The output of a final sigmoid activation is $g \in [0, 1]^H$, where $H = h + h/2 + \dots + INF$ denotes the total number of units of the NoTree layers. Separate chunks of g are used to modulate by feature gating the representations of NoTree: $[g_1, \dots, g_h]$ controls features at the first embedding, $[g_{h+1}, \dots, g_{h+h/2}]$ acts at the second layer, \dots , and so on, until exhausting $[g_{H-INF}, \dots, g_H]$ with the last layer prior the average pooling. In other words, g is used as a control mechanism on variables parameterization, gating their features via a learned tree-based signal. The resulting network (TreeGate) models the high-level idea that a branching scheme should *adapt to the tree evolution*, with variables’ selection criteria dynamically changing throughout the tree search.

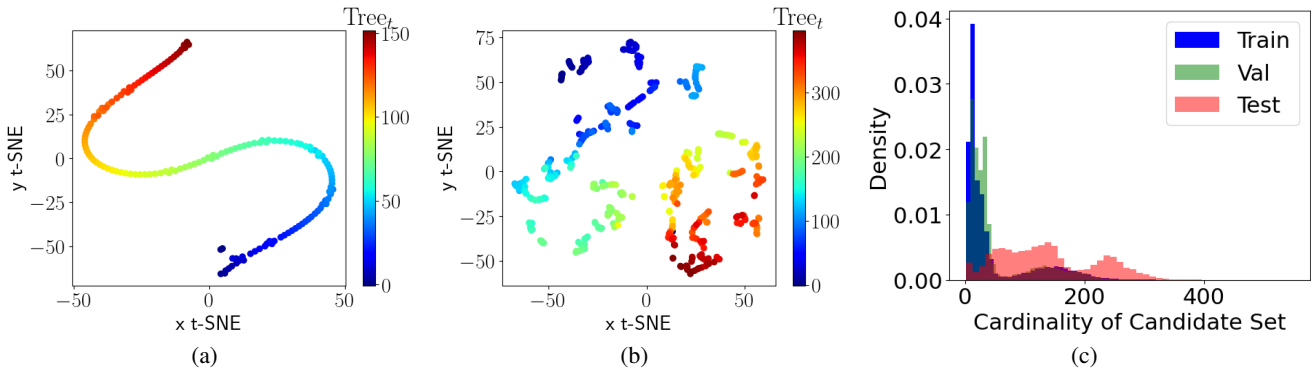


Figure 1: Evolution of $Tree_t$ throughout B&B as synthesized by t-SNE plots (perplexity=5), for instances (a) eil33-2 and (b) seymour1. (c) Histogram of $|C_t|$ in train, validation and test data.

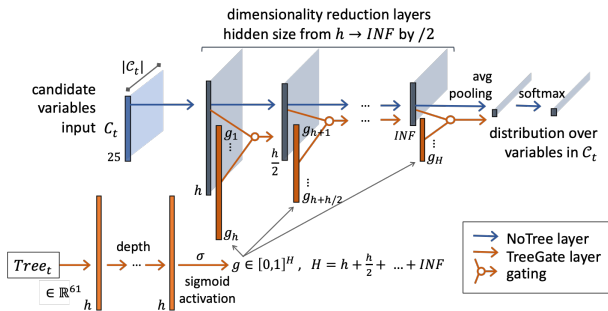


Figure 2: Candidate variables input C_t is processed by NoTree layers (in blue) to select a variable for branching. For the TreeGate model, the $Tree_t$ input is first embedded and then utilized in gating layers (in orange) on the candidates' representations.

3.3 Imitation Learning

We train our BVS policies via imitation learning, specifically behavioral cloning (Pomerleau 1991). Our expert branching scheme is SCIP default, *relpscost*, i.e., a reliability version of hybrid branching in which SB and PC scores are complemented with other ones reflecting the candidates' role in the search; *relpscost* is a more realistic expert (nobody uses SB in practice), and the most suited in our context, given the emphasis we put on the search tree. One of the main challenges of learning to imitate a complex BVS expert policy is that it is only possible to partially observe the true state of the solver. In our learning framework, we approximate the solver state with our parameterized B&B search tree state $x_t = \{C_t, Tree_t\}$ at branching step t . For each MILP instance, we roll-out SCIP to gather a collection of pairs $\mathcal{D} = (x_t, y_t)_{t=1}^N$ where $y_t \in C_t$ is the branching decision made by *relpscost* at branching step t . Our policies π_θ are trained to minimize the cross-entropy categorical loss function:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(x,y) \in \mathcal{D}} \log \pi_\theta(y | x). \quad (3)$$

3.4 Systematic Generalization

Our aim is to explore systematic generalization in learning to branch. We measure systematic generalization by evaluating how well trained policies perform on never-seen heterogeneous MILP instances. To begin, we remark that *relpscost* is a sophisticated ensemble of expertly-crafted heuristic rules that are dependent on high-level branching factors. Due to the systematic generalization failures currently plaguing even state-of-the-art DNNs (Lake and Baroni 2017; Johnson et al. 2017; Goodfellow, Shlens, and Szegedy 2015), we do not believe that current learning algorithms are capable of inferring these high-level branching factors from the raw MILP formulation data alone. Instead, by opting to train BVS policies in our hand-crafted input feature space: (1) the signal-to-noise ratio of the underlying branching factors is effectively increased, and (2) the likelihood of our models overfitting to superficial regularities is vastly decreased as this tree-oriented feature space is abstracted away from instance or class specific peculiarities of the MILP formulation. However, in order to achieve systematic generalization, inference of high-level latent factors must be paired with a corresponding composition mechanism. Here, the top-down neuro-modulatory prior of our TreeGate model represents a powerful mechanism for dynamically composing the tree state and the candidate variable states together. In summary, we hypothesize that BVS policies trained in our framework are able to better infer and compose latent higher order branching factors, which in turn enables flexible generalization across heterogeneous MILP instances.

4 Experiments

Our experiments are designed to carefully measure the inductive bias of BVS learning frameworks. We echo the sentiment of Melis, Kočíský, and Blunsom that merely scaling up a dataset is insufficient to explore the issue of systematic generalization in deep learning models, and we instead choose to curate a controlled dataset with the following properties:

- *Train/test split consisting of heterogeneous MILP instances.* This rules out the possibility of BVS policies

learning superficial class-specific branching rules and instead forces them to infer higher order branching factors shared across MILPs.

- *A restricted set of training instances.* While limiting the sample complexity of the training set poses a significant learning challenge, the learning framework with the best inductive bias will be the one that best learns to infer and compose higher order branching factors from such a difficult training set.

While we train our BVS policies to imitate the SCIP default policy `relpscost`, our objective is not to outperform `relpscost`. Indeed, expert BVS policies like `relpscost` are tuned over *thousands* of solvers’ proprietary instances: comprehensively improving on them is a very hard task – impossible to guarantee in our purely-research experimental setting – and should thus not be the only yardstick to determine the validity (and practicality) of a learned policy. Instead, we focus on evaluating how efficiently BVS learning frameworks can learn to infer and compose higher order branching factors by measuring generalization from a controlled training set to heterogeneous MILP test instances.

MILP dataset and solver setting In general, randomly-generated *generic* MILPs are too easy to be of interest; besides, public MILP libraries only contain few hundreds of instances, not all viable for our setting, and a careful dataset curation is thus needed. Comparisons of branching policies become clearer when the explored trees are manageable in size and the problems can be consistently solved to optimality. Thus we select 27 heterogeneous problems from real-world MILP benchmark libraries (Bixby et al. 1998; Koch et al. 2011; Gleixner et al. 2019; Mittelman 2020), focusing on instances whose tree exploration is on average relatively contained (in the tens/hundreds of thousands nodes, max.) and whose optimal value is known. We partition our selection into 19 train and 8 test problems, which are listed in Table 1(a) (see SM for more details).

We use SCIP 6.0.1. Modifying the solver configuration is common practice in BVS literature (Linderoth and Savelsbergh 1999), especially in a proof-of-concept setting in which our work is positioned. To reduce the effects of the other solver’s components on BVS, we work with a configuration specifically designed to fairly compare the performance of branching rules (Gamrath and Schubert 2018). In particular, we disable all primal heuristics and for each problem we provide the known optimal solution value as cutoff. We also enforce a time-limit of 1h. Further details on the solver parameters and hardware settings are reported in the SM.

Data collection and split We collect IL training data from SCIP roll-outs, gathering inputs $x_t = \{C_t, Tree_t\}$ and corresponding `relpscost` branching decisions (labels) $y_t \in C_t$. Given that each branching decision gives rise to a single data-point (x_t, y_t) , and that the search trees of the selected MILP instances are not extremely big, one needs to augment the data. We proceed in two ways.

TRAIN: air04, air05, demulti, eil33-2, istanbul-no-cutoff, l152lav, lseu, misc03, neos20, neos21, neos-476283, neos648910, pp08aCUTS, rmatr100-p10, rmatr100-p5, sp150x300d, stein27, swath1, vpm2

TEST: map18, mine-166-5, neos11, neos18, ns1830653, nu25-pr12, rail507, seymour1

(a)

	Total	(s, k) pairs
Train	85,533	$\{0, 1, 2, 3\} \times \{0, 1, 5, 10, 15\}$
Valid.	14,413	$\{4\} \times \{0, 1, 5, 10, 15\}$
Test	28,307	$\{0, 1, 2, 3, 4\} \times \{0\}$

(b)

Table 1: (a) List of MILP instances in train and test sets. (b) For train, validation and test set splits we report the total number of data-points and the seed- k pairs (s, k) from which they are obtained.

Policy	$h / d / LR$	Test acc (@5)	Val acc (@5)
NT	128 / - / 0.001	64.02 (88.51)	77.69 (95.88)
TG	64 / 5 / 0.01	83.70 (95.83)	84.33 (96.60)
GCNN	- / - / -	15.28 (44.16)	19.28 (38.44)

Table 2: Selected NoTree (NT) and TreeGate (TG) models with corresponding hyper-parameters, and test and validation accuracy. For GCNN we report *average* scores across 5 seeds; validation means use the best scores observed during training.

- We exploit MILPs *performance variability* (Lodi and Tramtani 2013), and obtain perturbed searches of the same instance by setting solver’s random seeds $s \in \{0, \dots, 4\}$ to control variables’ permutations.
- We diversify B&B explorations by running the branching scheme `random` for the first k nodes, before switching to SCIP default rule and starting data collection. The motivation behind this type of augmentation is to gather input states that are unlikely to be observed by an expert rule (He, Daume III, and Eisner 2014). We select $k \in \{0, 1, 5, 10, 15\}$, where $k = 0$ corresponds to a run without random branching. We apply this type of augmentation to *train instances only*.

One can quantify MILP variability by computing the coefficient of variation of the performance measurements (Koch et al. 2011); we report such scores and measure the effect of k initial random branchings in the SM. Overall, both (i) and (ii) appear effective to diversify our dataset. The final composition of train, validation and test sets is summarized in Table 1(b). Train and validation data come from the same instances; the test set contains samples from separate MILPs, using only type (i) augmentations.

An important measure to analyze the dataset is given by the size of the candidate sets (i.e., the varying dimensionality of the C_t inputs) contained in each split. Figure 1(c)

shows histograms for $|\mathcal{C}_t|$ in each subset. While in train and validation the candidate set sizes are mostly concentrated in the $[0, 50]$ range, the test set has a very different distribution of $|\mathcal{C}_t|$, and in particular one with a longer tail (over 300). In this sense, the test instances present never-seen branching data gathered from heterogeneous MILPs, and we test the generalization of our policies to *entirely unknown and larger branching sets*.

IL optimization and GCNN benchmark We train both IL policies using ADAM (Kingma and Ba 2015) with default $\beta_1 = 0.9, \beta_2 = 0.999$, and weight decay 1×10^{-5} . Our hyper-parameter search spans: learning rate $LR \in \{0.01, 0.001, 0.0001\}$, hidden size $h \in \{32, 64, 128, 256\}$, and depth $d \in \{2, 3, 5\}$. The factor by which units of NoTree are reduced is 2, and we fix $INF = 8$. We use PyTorch (Paszke et al. 2019) to train the models for 40 epochs, reducing LR by a factor of 10 at epochs 20 and 30. To benchmark our results, we also train the GCNN framework of Gasse et al. on our MILP dataset. Data collection and experiments are carried out as in Gasse et al., with full SB as expert, but we fix the solver setting as discussed above.

4.1 Results

In our context, standard IL metrics are informative yet incomplete measures of performance for a learned BVS model, and one also cares about assessing the policies’ behaviors when plugged in the solver environment. This is why in order to determine the best NoTree and TreeGate policies we take into account both types of evaluations. We first select few policies based on their test accuracy score; next, we specify them as custom branching rules in SCIP and perform full roll-outs on the entire MILP dataset, over five random seeds (i.e., 135 evaluations each). To summarize the policies’ performance in the solver, we compute the shifted geometric mean (with a shift of 100) of the total number of nodes, over the 135 B&B executions (ALL), and restricted to TRAIN and TEST instances.

Both types of metrics are extensively reported in the SM, together with the policies’ hyper-parameters. Incorporating an explicit parameterization of the state of the search tree to modulate BVS clearly aids generalization: the advantage of TreeGate over NoTree is evident in all metrics, and across multiple trained policies. What we observe is that best test accuracy does not necessarily translate into best solver performance. We select as best policies those yielding the best nodes average over the entire dataset (Table 2). In the case of TreeGate, the best model corresponds to that realizing the best top-1 test accuracy (83.70%), and brings a 19% (resp. 7%) improvement over the NoTree policy, in top-1 (resp. top-5) test accuracy. The GCNN models (trained, tested and evaluated over 5 seeds, as in Gasse et al.) struggle to fit data from heterogeneous instances: their average top-1 (resp. top-5) test accuracy is only 15.28% (resp. 44.16%), and across ALL instances they explore around three times the number of nodes needed by our policies. Note that GCNN is a memory intensive model, and we had to drastically reduce the batch size parameter to avoid memory issues when using our

instances. Learning curves and further details on training dynamics and test results can be found in the SM.

In solver evaluations, NoTree and TreeGate are also compared to SCIP default branching scheme `relpscost`, PC branching `pscost` and a random one. For `relpscost` we also compute the *fair* number of nodes (Gamrath and Schubert 2018), which accounts for those nodes that are processed as side-effects of SB-like explorations, specifically looking at domain reduction and cutoffs counts. In other words, the fair number distinguishes tree-size reductions due to better branching from those obtained by SB side-effects. For rules that do not involve any SB, the fair number and the usual nodes’ count coincide. The selected solver parametric setting (the same used for data collection and GCNN benchmark) allows a meaningful computation of the fair number of nodes, and a honest comparison of branching schemes.

Both NoTree and TreeGate policies are able to solve all instances within the 1h time-limit, like `relpscost`. In contrast, GCNN hits the limit on 7 instances (24 times in total), while `random` does so on 4 instances (17 times in total) and `pscost` on one instance only (neos18), a single time. Table 3 reports the nodes’ means for every test instance over five runs (see SM for complete instance-specific results), as well as measures aggregated over train and test sets, and the entire dataset. In aggregation, TreeGate is always better than NoTree, the former exploring on average trees with 14.9% less nodes. This gap becomes more pronounced when measured over test instances only (27%), indicating the advantage of TreeGate over NoTree when exploring unseen data. Results are less clear-cut from an instance-wise perspective, with neither policy emerging as an absolute winner, though the reductions in tree sizes achieved by TreeGate are overall more pronounced. While the multiple time-limits of GCNN hinder a proper comparison in terms of explored nodes, results clearly indicate that the difficulties of GCNN exacerbate over never-seen, heterogeneous test instances.

Our policies also compare well to other branching rules: both NoTree and TreeGate are substantially better than `random` across all instances, and always better than `pscost` in aggregated measures. Only on one training instance both policies are much worse than `pscost` (neos-476283); in the test set, GCNN appears competitive with our models only on neos11. As expected, `relpscost` still realizes the smallest trees, but on 11 (out of 27) instances at least one among NoTree and TreeGate explores less nodes than the `relpscost` fair number. In general, our policies realize tree sizes comparable to the SCIP ones, when SB side effects are taken into account.

5 Related Work

Among the first attempts in “learning to branch”, Alvarez, Louveaux, and Wehenkel perform regression to learn proxies of SB scores. Instead, Khalil et al. propose to learn the ranking associated with such scores, and train instance-specific models (that are not end-to-end policies) via SVM^{rank}. Also Hansknecht, Joormann, and Stiller treat BVS as a ranking problem, and specialize their models to the combinatorial class of time-dependent traveling sales-

Instance	NoTree	TreeGate	GCNN	random	pscost	relpscost (fair)
ALL	1241.79	1056.79	*3660.32	*6580.79	*1471.61	286.15 (719.20)
TRAIN	834.40	759.94	*1391.41	*2516.04	884.37	182.27 (558.34)
TEST	3068.96	2239.47	*33713.63	*61828.29	*4674.34	712.77 (1276.76)
map18	457.89	575.92	*3907.64	11655.33	1025.74	270.25 (441.18)
mine-166-5	3438.44	4996.48	*233142.25	*389437.62	4190.41	175.10 (600.22)
neos11	3326.32	3223.46	1642.07	29949.69	4728.49	2618.27 (5468.05)
neos18	15611.63	10373.80	40794.74	228715.62	*133437.40	2439.29 (5774.36)
ns1830653	6422.37	5812.03	*22931.45	288489.30	12307.90	3489.07 (4311.84)
nu25-pr12	357.00	86.80	*45982.34	1658.41	342.47	21.39 (105.61)
rail507	9623.05	3779.05	*75663.48	*80575.84	4259.98	543.39 (859.37)
seymour1	3202.20	1646.82	*319046.04	*167725.65	3521.47	866.32 (1096.67)

Table 3: Total number of nodes explored by learned and SCIP policies for test instances and aggregated over sets, in shifted geometric means over 5 runs on seeds $\{0, \dots, 4\}$. We mark with * the cases in which time-limits were hit. For *relpscost*, we also compute the *fair* number of nodes.

man problems. More recently, the work of Balcan et al. learns mixtures of existing branching schemes for different classes of synthetic problems, focusing on sample complexity guarantees. In Di Liberto et al., a portfolio approach to BVS is explored. Similarly to us, Gasse et al. frames BVS as classification of SB-expert branching decisions and employs a GCNN model to learn branching. Proposed features in Gasse et al. (as in Alvarez, Louveaux, and Wehenkel; Khalil et al.) focus on static, parameters-dependent properties of MILPs and node LP relaxations, whereas our representations aim at capturing the temporality and dynamism of BVS. Although their resulting policies are specializations of SB that appear to effectively capture structural characteristics of some classes of combinatorial optimization problems, and are able to generalize to larger formulations from the same distribution, we showed how such policies fail to attain a broader generalization paradigm.

Still concerning the B&B framework, He, Daume III, and Eisner employ IL to learn a heuristic class-specific node selection policy; Song et al. propose instead a retrospective approach on IL. A reinforcement learning (RL) approach for node selection can be found in Sabharwal, Samulowitz, and Reddy, where a Multi-Armed Bandit is used to model the tree search.

Feature gating has a long and successful history in machine learning (see Makkuva et al.), ranging from LSTMs (Hochreiter and Schmidhuber 1997) to GRUs (Chung et al. 2014). The idea of using a tree state to drive a feature gating of the branching variables is an example of top-down modulation, which has been shown to perform well in other deep learning applications (Shrivastava et al. 2016; Lin et al. 2016; Son and Mishra 2018). With respect to learning across non-static action spaces, the most similar to our work is Chandak et al., in the continual learning setting. Unlike the traditional Markov Decision Process formulation of RL, the input to our policies is not a generic state but rather includes a parameterized hand-crafted representation of the available actions, thus continual learning is not a relevant concern for our framework. Other works from the RL setting learn representations of *static* action spaces (Dulac-Arnold et al. 2015; Chandak et al. 2019a), while in contrast the action space of

BVS changes dynamically with $|\mathcal{C}_t|$.

6 Conclusions and Future Directions

Branching variable selection is a crucial factor in B&B success, and we setup a novel imitation learning framework to address it. We sought to learn branching policies that generalize across heterogeneous MILPs, regardless of the instances’ structure and formulation size. In doing so, we undertook a step towards a broader type of generalization. The novelty of our approach is relevant for both the ML and the MILP worlds. On the one hand, we developed parameterizations of the candidate variables and of the search trees, and designed a DNN architecture that handles candidate sets of varying size. On the other hand, the data encoded in our $Tree_t$ parameterization is not currently exploited by state-of-the-art MILP solvers, but we showed that this type of information could indeed help in adapting the branching criteria to different search dynamics. Our results on MILP benchmark instances clearly demonstrated the advantage of incorporating a search-tree context to modulate BVS and aid generalization to heterogeneous problems, in terms of both better test accuracy and smaller explored B&B trees. The comparison with the GCNN setup of Gasse et al. reinforced our conclusions: experiments showcased the inability of the GCNN paradigm alone to generalize to new instances for which no analogs were available during training. One crucial step towards improving over state-of-the-art solvers is precisely that of being able to generalize across heterogeneous problems, and our work is the first paper in the literature attaining this target.

There surely are additional improvements to be gained by continuing to experiment with IL methods for branching, and also by exploring innovative RL settings. Indeed, the idea and the benefits of using an explicit parameterization of B&B search trees – which we demonstrated in the IL setup – could be expanded even more in the RL one, for both state representations and the design of branching rewards.

Acknowledgements

We would like to thank Ambros Gleixner, Gerald Gamrath, Laurent Charlin, Didier Chételat, Maxime Gasse, Antoine Prouvost, Leo Henri and Sébastien Lachapelle for helpful discussions on the branching framework. We also thank Compute Canada for compute resources. This work was supported by CIFAR and IVADO. We also thank the anonymous reviewers for their helpful feedback.

References

- Achterberg, T.; and Berthold, T. 2009. *Hybrid Branching*, 309–311. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-01929-6_23.
- Achterberg, T.; Koch, T.; and Martin, A. 2005. Branching rules revisited. *Oper Res Lett* 33(1): 42–54. doi:10.1016/j.orl.2004.04.002.
- Achterberg, T.; and Wunderling, R. 2013. *Mixed Integer Programming: Analyzing 12 Years of Progress*, 449–481. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-38189-8_18.
- Alvarez, M. A.; Louveaux, Q.; and Wehenkel, L. 2017. A Machine Learning-Based Approximation of Strong Branching. *INFORMS Journal on Computing* 29(1): 185–195. doi:10.1287/ijoc.2016.0723.
- Applegate, D.; Bixby, R.; Chvatal, V.; and Cook, B. 1995. Finding Cuts in the TSP (A Preliminary Report). Technical report.
- Balcan, M.-F.; Dick, T.; Sandholm, T.; and Vitercik, E. 2018. Learning to Branch. In *International Conference on Machine Learning*, 344–353.
- Bengio, Y.; Lodi, A.; and Prouvost, A. 2018. Machine Learning for Combinatorial Optimization: a Methodological Tour d’Horizon. *arXiv:1811.06128*.
- Benichou, M.; Gauthier, J.; Girodet, P.; and Hentges, G. 1971. Experiments in mixed-integer programming. *Math Program* 1: 76–94.
- Berthold, T.; Hendel, G.; and Koch, T. 2017. From feasibility to improvement to proof: three phases of solving mixed-integer programs. *Optimization Methods and Software* 33(3): 499 – 517. doi:10.1080/10556788.2017.1392519.
- Bixby, R. E.; Ceria, S.; McZeal, C. M.; and Savelsbergh, M. W. 1998. An updated mixed-integer programming library: MIPLIB 3. Technical Report TR98-03. URL <http://miplib2010.zib.de/miplib3/miplib.html>. Accessed: 2019.
- Chandak, Y.; Theocharous, G.; Kostas, J.; Jordan, S.; and Thomas, P. 2019a. Learning Action Representations for Reinforcement Learning. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, 941–950. PMLR.
- Chandak, Y.; Theocharous, G.; Nota, C.; and Thomas, P. S. 2019b. Lifelong Learning with a Changing Action Set. *CoRR* abs/1906.01770.
- Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555*.
- Di Liberto, G.; Kadioglu, S.; Leo, K.; and Malitsky, Y. 2016. DASH: Dynamic Approach for Switching Heuristics. *Eur J Oper Res* 248(3): 943–953. doi:10.1016/j.ejor.2015.08.018.
- Dulac-Arnold, G.; Evans, R.; Sunehag, P.; and Coppin, B. 2015. Reinforcement Learning in Large Discrete Action Spaces. *CoRR* abs/1512.07679.
- Fischetti, M.; Lodi, A.; and Zarpellon, G. 2019. Learning MILP Resolution Outcomes Before Reaching Time-Limit. In Rousseau, L.-M.; and Stergiou, K., eds., *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 275–291. Cham: Springer International Publishing.
- Gamrath, G.; and Schubert, C. 2018. Measuring the Impact of Branching Rules for Mixed-Integer Programming. In Klierer, N.; Ehmke, J. F.; and Borndörfer, R., eds., *Operations Research Proceedings 2017*, 165–170. Cham: Springer International Publishing.
- Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d’Alché Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 32*, 15554–15566. Curran Associates, Inc.
- Gleixner, A.; Bastubbe, M.; Eifler, L.; Gally, T.; Gamrath, G.; Gottwald, R. L.; Hendel, G.; Hojny, C.; Koch, T.; Lübbecke, M.; Maher, S. J.; Miltenberger, M.; Müller, B.; Pfetsch, M.; Puchert, C.; Rehfeldt, D.; Schlösser, F.; Schubert, C.; Serrano, F.; Shinano, Y.; Viernickel, J. M.; Walter, M.; Wegscheider, F.; Witt, J. T.; and Witzig, J. 2018. The SCIP Optimization Suite 6.0. Technical Report 18-26, ZIB, Takustr. 7, 14195 Berlin.
- Gleixner, A.; Hendel, G.; Gamrath, G.; Achterberg, T.; Bastubbe, M.; Berthold, T.; Christophel, P. M.; Jarck, K.; Koch, T.; Linderoth, J.; Lübbecke, M.; Mittelman, H. D.; Ozyurt, D.; Ralphs, T. K.; Salvagnin, D.; and Shinano, Y. 2019. *MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library*. URL http://www.optimization-online.org/DB_HTML/2019/07/7285.html. Accessed: 2019.
- Goodfellow, I.; Shlens, J.; and Szegedy, C. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*.
- Hansknecht, C.; Joormann, I.; and Stiller, S. 2018. Cuts, Primal Heuristics, and Learning to Branch for the Time-Dependent Traveling Salesman Problem. *arXiv:1805.01415*.
- He, H.; Daume III, H.; and Eisner, J. M. 2014. Learning to Search in Branch and Bound Algorithms. In Ghahramani, Z.; Welling, M.; Cortes, C.; Lawrence, N. D.; and Weinberger, K. Q., eds., *Advances in Neural Information Processing Systems 27*, 3293–3301. Curran Associates, Inc.
- Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8): 1735–1780.
- Johnson, J.; Hariharan, B.; van der Maaten, L.; Fei-Fei, L.; Zitnick, C. L.; and Girshick, R. 2017. CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning. In *CVPR*.
- Khalil, E. B.; Bodic, P. L.; Song, L.; Nemhauser, G.; and Dilkina, B. 2016. Learning to Branch in Mixed Integer Programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI’16*, 724–731. AAAI Press.
- Kingma, D. P.; and Ba, J. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.
- Koch, T.; Achterberg, T.; Andersen, E.; Bastert, O.; Berthold, T.; Bixby, R. E.; Danna, E.; Gamrath, G.; Gleixner, A. M.; Heinz, S.; Lodi, A.; Mittelman, H.; Ralphs, T.; Salvagnin, D.; Steffy, D. E.;

- and Wolter, K. 2011. MIPLIB 2010. *Mathematical Programming Computation* 3(2): 103. doi:10.1007/s12532-011-0025-9.
- Lake, B. M.; and Baroni, M. 2017. Still not systematic after all these years: On the compositional skills of sequence-to-sequence recurrent networks. *CoRR* abs/1711.00350.
- Land, A. H.; and Doig, A. G. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica* 28(3): 497–520.
- Le Bodic, P.; and Nemhauser, G. 2017. An abstract model for branching and its application to mixed integer programming. *Math Program* 1–37. doi:10.1007/s10107-016-1101-8.
- Lin, T.; Dollár, P.; Girshick, R. B.; He, K.; Hariharan, B.; and Belongie, S. J. 2016. Feature Pyramid Networks for Object Detection. *CoRR* abs/1612.03144.
- Linderoth, J. T.; and Savelsbergh, M. W. P. 1999. A Computational Study of Search Strategies for Mixed Integer Programming. *INFORMS Journal on Computing* 11(2): 173–187. doi:10.1287/ijoc.11.2.173.
- Lipton, R. J.; and Regan, R. W. 2012. Branch And Bound—Why Does It Work? <https://rjlipton.wordpress.com/2012/12/19/branch-and-bound-why-does-it-work/>. Accessed: 2019.
- Lodi, A. 2009. Mixed integer programming computation. In Jünger, M.; Liebling, T.; Naddef, D.; Nemhauser, G.; Pulleyblank, W.; Reinelt, G.; Rinaldi, G.; and Wolsey, L., eds., *50 Years of Integer Programming 1958-2008*, 619–645. Springer Berlin Heidelberg.
- Lodi, A.; and Tramontani, A. 2013. *Performance Variability in Mixed-Integer Programming*, chapter 1, 1–12. INFORMS. doi:10.1287/educ.2013.0112.
- Lodi, A.; and Zarpellon, G. 2017. On learning and branching: a survey. *TOP* 25(2): 207–236. doi:10.1007/s11750-017-0451-6.
- Maher, S.; Miltenberger, M.; Pedroso, J. P.; Rehfeldt, D.; Schwarz, R.; and Serrano, F. 2016. PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite. In Greuel, G.-M.; Koch, T.; Paule, P.; and Sommese, A., eds., *Mathematical Software – ICMS 2016*, 301–307. Cham: Springer International Publishing. ISBN 978-3-319-42432-3.
- Makkuva, A. V.; Oh, S.; Kannan, S.; and Viswanath, P. 2019. Learning in Gated Neural Networks. *CoRR* abs/1906.02777.
- Melis, G.; Kočíský, T.; and Blunsom, P. 2020. Mogrifier LSTM. In *International Conference on Learning Representations*.
- Mittelmann, H. D. 2020. MILPlib. <http://plato.asu.edu/ftp/milp/>. Accessed 2019.
- Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems* 32, 8024–8035. Curran Associates, Inc.
- Pomerleau, D. A. 1991. Efficient Training of Artificial Neural Networks for Autonomous Navigation. *Neural Computation* 3(1): 88–97.
- relpscost. 2019. Code for the relpscost branching rule in SCIP. https://scip.zib.de/doc-6.0.0/html/branch_relpscost_8c_source.php#l00524. Accessed: 2019.
- Sabharwal, A.; Samulowitz, H.; and Reddy, C. 2012. Guiding Combinatorial Optimization with UCT. In Beldiceanu, N.; Jussien, N.; and Pinson, É., eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28 – June 1, 2012. Proceedings*, Lecture Notes in Computer Science, 356–361. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-29828-8_23.
- Shrivastava, A.; Sukthankar, R.; Malik, J.; and Gupta, A. 2016. Beyond Skip Connections: Top-Down Modulation for Object Detection. *CoRR* abs/1612.06851.
- Son, J.; and Mishra, A. K. 2018. ExGate: Externally Controlled Gating for Feature-based Attention in Artificial Neural Networks. *CoRR* abs/1811.03403.
- Song, J.; Lanka, R.; Zhao, A.; Bhatnagar, A.; Yue, Y.; and Ono, M. 2018. Learning to Search via Retrospective Imitation. *arXiv preprint 1804.00846*.
- van der Maaten, L.; and Hinton, G. 2008. Visualizing High-Dimensional Data Using t-SNE. *Journal of Machine Learning Research* 9(nov): 2579–2605. Pagination: 27.
- Wolsey, L. A. 1998. *Integer programming*. New York, NY, USA: Wiley-Interscience.