Learning Generalized Strong Branching for Set Covering, Set Packing, and 0-1 Knapsack Problems

Yu Yang*[1], Bistra Dilkina[3], Natashia Boland[2], and Martin Savelsbergh[2]

[1]*Department of Industrial and Systems Engineering, University of Florida, 303 Weil Hall, Gainesville, FL 32603*
[2]*H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, 755 Ferst Dr NW, Atlanta, GA 30332*
[3]*Viterbi School of Engineering, University of Southern, 941 Bloom Walk, Los Angeles, CA 90089*
yu.yang@ise.ufl.edu, dilkina@usc.edu
{natashia.boland, martin.savelsbergh}@isye.gatech.edu

Abstract

Branching on a set of variables, rather than on a single variable, can give tighter bounds at the child nodes and can result in smaller search trees. However, selecting a good set of variables to branch on is even more challenging than selecting a good single variable to branch on. Generalized strong branching extends the strong branching concepts developed for choosing a single variable to choosing a set of variables. As the computational requirements of a full implementation of strong branching are prohibitive, we use extreme gradient boosting to train a model to predict the ranking of (sets of) candidate variables. An extensive computational study using instances from three well-known classes of optimization problems demonstrates that branching on sets of variables outperforms branching on a single variable, that a learned model can be used effectively to select among (sets of) candidate variables, and that the learned strong branching strategies outperform the default branching strategy of state-of-the-art commercial solver CPLEX in terms of both the number of nodes explored in the search tree and the time it takes to explore the search tree.

Keywords— branch and bound, machine learning, binary optimization

1 Introduction

Mixed integer programming (MIP) is a powerful mathematical modeling framework that has been applied in various domains, e.g., airline crew scheduling, service network design, online advertising, and cloud

*Corresponding author.

computing. Because of the tremendous algorithmic advances in state-of-the-art commercial MIP solvers, such as CPLEX and Gurobi, many practically relevant instances can be solved efficiently and reliably. However, there remain settings, for example operational settings in which the time available to make decisions is limited, where the use of MIP technology is not yet a viable option because instances still take too long to solve. This motivates research into techniques that can accelerate the solution of MIPs. Our research falls in this category and investigates whether machine learning (ML) techniques can be used to achieve speedups in solution time.

An effective branching strategy is critically important for modern MIP solvers, such as SCIP, CPLEX, and Gurobi. A computational study in Achterberg and Wunderling (2013), using CPLEX 12.5, shows that a naive branching scheme, i.e., branch on the most infeasible variable, increases computation time by a factor of 4.5 and number of nodes explored by a factor of 6.5 times compared to the default branching scheme. Therefore, it is not surprising that branching strategies, especially the choice of a (single) branching variable, has received much attention over the years, e.g., pseudo-cost branching (PB) (Bénichou et al. 1971), strong branching (SB) (Applegate et al. 1995), hybrid strong and pseudo-cost branching (HB) (Gauthier and Ribière 1977, Linderoth and Savelsbergh 1999), reliability branching (RB) (Achterberg et al. 2005), backdoor branching (Fischetti and Monaci 2011) . We refer the interested reader to Achterberg (2007) for a more in-depth review of branching strategies. Branching on sets of variables has also been explored, but mostly in the context of a specific problem structure or with a specific set of variables, e.g., special ordered set branching (Beale and Tomlin 1970, Beale 1979), constraint branching (Ryan and Foster 1981), explicit-constraint branching (Appleget and Wood 2000), and hyperplane branching (Pataki and Tural 2010).

We investigate a multi-variable or set branching strategy that does not rely on a specific problem structure or specific set of variables. Specifically, for an arbitrary set S of indices of integer variables with $\sum_{i \in S} x_i^{LP} = f$, where x^{LP} is the solution of the linear programming (LP) relaxation at the current node in the search tree, and f is fractional, we create two branches where $\sum_{i \in S} x_i \geq \lceil f \rceil$ is imposed in the *up* branch and $\sum_{i \in S} x_i \leq \lfloor f \rfloor$ is imposed in the *down* branch. This is a natural generalization of single-variable dichotomy branching, which is recovered when S is restricted to be a singleton. The computational study in Yang et al. (2021) demonstrates the potential of this set branching scheme for 0-1 knapsack problems. It is worth noting that we (implicitly) assume that there are at least three variables in an instance. Therefore, when $x_1^{LP} = x_2^{LP} = 0.5$, there is at least one other variable x_3. If x_3^{LP} is fractional, then we can pick the set $S = \{1, 2, 3\}$. If x_3^{LP} is integer, then we can pick $S = \{1, 3\}$ or $S = \{2, 3\}$. In case there are only two variables, we can pick $S = \{1\}$ or $S = \{2\}$, which reduces to the standard single-variable branching.

The primary challenge associated with the use of a set branching strategy is the efficient selection of an effective branching set. Almost all single-variable branching schemes, pure pseudo-cost branching being the exception, in one way or another rely on strong branching concepts. Strong branching (SB) is a look-ahead strategy in which for each fractional variable in the LP relaxation at the current node in the search tree, the optimal LP relaxation values of the up and the down branch are computed and converted into a SB score, which is subsequently used to select the fractional variable to branch on. Strong branching has been shown to result in small search trees, but is computationally prohibitive. To reduce the computational burden, only a subset of fractional variables is considered, or the SB score for a variable is computed only the first time the variable is fractional and pseudo-costs are used afterwards, or the SB score for a variable is computed the first time it is fractional and whenever the pseudo-cost is no longer deemed reliable. SB scores can be computed for sets of variables as well. We will use *generalized strong branching* (GSB) to refer to using SB scores to select a set of variables to branch on. If only sets of variables of up to size k are considered, this will be denoted as GSB-k.

The computational challenges associated with computing SB scores for sets of variables are even greater, as the number of sets of variables with a fractional value in the LP relaxation at the current node is likely to be much larger than the number of variables with a fractional values in the LP relaxation. Furthermore, the number of sets of size less than or equal to k grows exponentially with k, which makes GSB-k impractical for all but very small values of k. Even for $k = 2$, straightforward approaches to limiting the time spent computing SB scores are insufficient to lead to a viable implementation of GSB-2. Therefore, we devise an effective strategy to limit the number of variable sets considered and exploit advances in ML to (efficiently) select a good set to branch on.

The use of ML techniques to mimic the behavior of SB is not new. The idea is to solve many instances offline using a strong branching strategy, and collect information throughout the solution process, e.g., features of candidate branching variables, SB scores, ranking of SB scores at a node, and then train a ML model to predict SB scores or a ranking of candidate variables based on their features. The trained model is then applied in an online fashion to select the variables to branch on. Such an offline-online approach has been used in other contexts as well, see for example Jiménez-Cordero et al. (2021), Bertsimas and Stellato (2021). The promising computational results reported in papers such as Àlvarez et al. (2014, 2016), Khalil et al. (2016), Gasse et al. (2019) (see Section 2) suggest that by carefully choosing a ML model for carefully selected set of features a branching strategy can be developed that is effective (mimics SB) and efficient (does not require solving LPs to compute SB scores).

We explore similar ideas, but focus on mimicking GSB-2, as it can yield search trees that are significantly smaller than those produced with SB. More specifically, we focus on learning GSB-2 for three well-known classes of integer programs (IPs): set covering, set packing, and 0-1 knapsack. As mentioned above, practical implementations of SB only compute SB scores for a small set of candidates. Our strategy for limiting the set of candidate pairs in GSB-2 is to consider only pairs that include the variable identified as the best single variable to branch on. Thus, our approach relies on two learned models: one to select the best single variable to branch on from among a small set of candidate variables, and one to select the best pair of variables to branch on from among a small set of candidate pairs of variables.

We employ a systematic approach to identify the features to be used to train the ML model. We start from 25 features that we believe, based on our domain knowledge and on insights in the published literature discussed above, provide meaningful information for predicting the ranking of candidates by SB scores. After training data has been gathered for a particular problem class, a feature selection procedure is applied to eliminate features that are insignificant for that class, and to reduce the computational burden of variable set selection. The result is a small set of significant features (about 5) that can be computed efficiently. The learned model based on this small set of features is thus dependent on the problem class, and so can exploit problem class structure to arrive at a computationally effective approach.

The main contributions of our research are summarized as follows.

- We show that branching on sets of variables, as proposed in Yang et al. (2021), can be implemented effectively by carefully limiting the sets of variables to consider; multi-variable GSB outperforms single-variable SB in terms of the size of branch-and-bound search trees for three classes of well-known integer programs.
- We demonstrate that both SB and GSB can be enhanced by a ML technique, i.e., by learning to rank variables via gradient boosting. We show, for the first time, that learned branching strategies can outperform the default branching strategy of the state-of-the-art commercial solver CPLEX in terms of both the number of nodes explored in the search tree and the time taken to explore it. Furthermore, the learned GSB mostly outperforms the learned SB, which indicates that the

benefits of branching on more than one variable can be retained in the ML process.

The remainder of the paper is organized as follows. In Section 2, we review recent attempts on leveraging ML to enhance decisions made inside MIP solvers. In Section 3, we introduce GSB-k and our implementation of GSB-2. In Section 4, we present our methodology for learning SB and extend this framework to learn GSB-2. Section 5 presents the results of an extensive computational study. We finish, in Section 6, with final remarks.

2 Literature Review

Combinatorial optimization has been applied to tackle decision problems involving discrete variables that lie at the heart of some ML techniques (see Gambella et al. 2021). At the same time, there have been many attempts to leverage ML techniques to accelerate the solution of combinatorial optimization problems (see Lodi and Zarpellon 2017, Bengio et al. 2021, Karimi-Mamaghan et al. 2021). In particular, it has been shown that important decisions inside MIP solvers, such as selecting primal heuristics, search strategies, branching variable selection, and cutting planes can be done more effectively using ML techniques.

Khalil et al. (2017) apply logistic regression to learn a binary classification model that predicts whether a heuristic will succeed at a given node of the search tree. Their numerical study demonstrates that the performance of one of the fastest open-source solvers, SCIP can be improved by up to 6% on a set of benchmark instances, and by up to 60% on a family of hard Independent Set instances. Chmiela et al. (2021) propose a data-driven framework for scheduling heuristics in an exact MIP solver. Compared to the default settings of SCIP, their learned scheduling method can reduce the average primal integral by up to 49% on a class of challenging instances.

In He et al. (2014), an adaptive node selection strategy is learned by imitation learning for four different classes of problems. The learned node selection strategy is able to significantly enhance the performance of SCIP. Yilmaz and Yorke-Smith (2021) use imitation learning but focus on deciding which of a node's child node to select. Empirical results on five MIP datasets indicate that their method significantly outperforms the state-of-the-art precedent in the literature but fails to beat SCIP.

A deep reinforcement learning (RL) method is developed for intelligent adaptive selection of cutting planes in Tang et al. (2020). Their numerical study across a wide range of benchmark problems shows that their trained RL agent significantly outperforms human-designed heuristics. Huang et al. (2021) propose a data-driven cut selection approach, which is demonstrated to be effective and generalizes better across multiple problems with different properties compared to commonly used heuristics for cut selection.

In Àlvarez et al. (2014) extremely randomized forests are used to train a ML model of SB offline. Their computational study shows that the learned model is faster than full SB, but that it is outperformed by RB. Àlvarez et al. (2016) use online learning (using simple linear regression) to predict SB scores. The authors observe improvement over (their implementation of) full SB and RB, but it is difficult to tell from the performance profile how significant the improvement is. Khalil et al. (2016) propose an online ML model trained by SVM$^{\text{rank}}$ to predict a ranking of candidate variables by SB scores. Their computational study shows that using the ML model has some benefits over using PB, but cannot compete with the default branching scheme of CPLEX. Gasse et al. (2019) use imitation learning (using a graph convolutional neural network) to learn SB and report that the resulting branching scheme outperforms the default branching scheme of SCIP on three classes of optimization problems.

3 Generalized Strong Branching

GSB refers to a branching strategy in which set branching is employed and in which the set of variables to branch on is determined using the SB score. To obtain the SB score for a candidate set S at the current node in the search tree, two LPs have to be solved: one in which the branching constraint $\sum_{i \in S} x_i \geq \lceil f \rceil$ is added (the up branch) and one in which the branching constraint $\sum_{i \in S} x_i \leq \lfloor f \rfloor$ is added (the down branch). The SB score is computed using a score function $\mathit{score}(\Delta^-, \Delta^+)$, where $\Delta^- = |z - z^-|$, $\Delta^+ = |z - z^+|$, and z, z^+, and z^- are the objective values of the LP relaxations at the current node, the up branch, and the down branch, respectively. The candidate set S with the largest SB score is selected to branch on. If no restrictions are imposed on the candidate sets S, GSB becomes computationally intractable as the number of sets of size k is $\mathcal{O}(n^k)$. To explore the benefits of GSB, we therefore only consider GSB-2, i.e., branching on sets of size up to 2.

3.1 Score Function

Achterberg (2007) shows that the score function used can significantly impact the performance of SB. His computational study shows that

$$\mathit{score}(\Delta^-, \Delta^+) = \max\{\Delta^-, \epsilon\} \cdot \max\{\Delta^+, \epsilon\}$$

with $\epsilon = 10^{-6}$, the product form, outperforms

$$\mathit{score}(\Delta^-, \Delta^+) = (1 - \mu) \min\{\Delta^-, \Delta^+\} + \mu \max\{\Delta^-, \Delta^+\},$$

the more traditional sum form, by almost 15%. The reason, most likely, is that the product form better balances the contribution of the two branches than the sum form. When one of Δ^- and Δ^+ is much larger than the other, the SB score is dominated by the larger one in the sum form, which implies that the SB score does not fully capture the information provided by the two branches. On the other hand, with the product form, even a small value affects the SB score as it pushes down the value of the product.

Even though the product form of the score function outperforms the sum form of the score function, we believe it should be slightly modified when used for set branching. Consider, for example, set covering problems, where all constraints have the form $\sum_{k \in K} x_k \geq 1$. When branching on a single variable, the effect of fixing a variable to zero or fixing a variable to one is somewhat similar because no further variables can be fixed directly. As result, the search tree is likely somewhat balanced. When branching on a pair of variables, this is no longer the case. There is a difference between adding $x_i + x_j \leq 0$, which fixes two variables, and adding $x_i + x_j \geq 1$, which fixes no variables, and less balanced search trees may result. A simple product does not capture these differences. By placing more importance on the branch with more freedom a more effective SB score may be obtained. Therefore, we use a weighted product score function

$$\max\{\underline{z}/z - 1.0,\ \epsilon\}^{\alpha} \cdot \max\{\overline{z}/z - 1.0,\ \epsilon\}^{\beta}, \tag{1}$$

where $\underline{z}$ is the LP value of the branch with more freedom, $\overline{z}$ is that of the other branch, $\alpha > 1$, $\beta < 1$, $\alpha + \beta = 2$ and $\epsilon = 10^{-6}$.

We derive our score function from the following standard product form:

$$\mathit{score}(\Delta^-, \Delta^+) = \max\{\Delta^-, \epsilon\} \cdot \max\{\Delta^+, \epsilon\},$$

where $\Delta^- = |z - z^-|$, $\Delta^+ = |z - z^+|$, and z, z^+, and z^- are the objective values of the LP relaxations at the current node, the up branch, and the down branch, respectively. For minimization problem, we have $z^- \geq z$ and $z^+ \geq z$ and by convention, an infeasible node has objective value $+\infty$ (or equivalently, a very large number). Thus, $\Delta^- = z^- - z$, $\Delta^+ = z^+ - z$, and

$$\mathit{score} = \max\{z^- - z, \epsilon\} \cdot \max\{z^+ - z, \epsilon\} = \max\{\underline{z} - z, \epsilon\} \cdot \max\{\overline{z} - z, \epsilon\},$$

where $\underline{z}$ is the LP value of the branch with more freedom, which can be either z^- or z^+, and $\overline{z}$ is that of the other branch. If we normalize by dividing the above score by z^2, then it reduces to

$$\begin{aligned}\mathit{score} &= \max\{\underline{z}/z - 1.0, \epsilon/z\} \cdot \max\{\overline{z}/z - 1.0, \epsilon/z\} \\ &\approx \max\{\underline{z}/z - 1.0, \epsilon\} \cdot \max\{\overline{z}/z - 1.0, \epsilon\} \\ &= \max\{\underline{z}/z - 1.0, \epsilon\}^{\alpha} \cdot \max\{\overline{z}/z - 1.0, \epsilon\}^{\beta},\end{aligned}$$

where $\alpha = \beta = 1$, and thus $\alpha + \beta = 2$. Essentially, instead of using the changes in the absolute value, we use relative changes. In our score function, we further take into account the degree of freedom of the two branches, and the one with more freedom will have a higher power $\alpha > 1$. The power of the other branch will be set accordingly to $\beta = 2 - \alpha < 1$. The restriction that $\alpha + \beta = 2$ is not dependent on the number of variables considered. It is decided by the number of new branches created each time, or equivalently, the number of items in the product for computing the score. Thus if three variables are considered, we still use $\alpha + \beta = 2$.

The results of a small computational experiment using 100 randomly generated set covering instances, in which we compare the performance of the weighted product score function (with $\alpha > 1$ and $\beta < 1$) to the original product score function (with $\alpha = 1$ and $\beta = 1$) for different values of α and β revealed that the weighted product score function is not very sensitive to the values of α and β and performs well as long as $\alpha > 1$ and $\beta < 1$. Therefore, in all further experiments, we use $\alpha = 1.5$ and $\beta = 0.5$. The benefit of using this weighted produce score function is illustrated in Figure 1, where we show the number of nodes in the search tree for both the original and the weighted product score function for the 100 instances (in nondecreasing order of nodes in the search tree when using the weighted product function). On average the weighted product score reduces the number of nodes in the search tree by about 20%.

3.2 Variable Selection

When the number of variables, n, is large, considering every possible set of size up to 2 will be computationally prohibitive. A straightforward way to reduce the number of candidates is to randomly select a small number of qualifying variable pairs, i.e., a pair (x_i, x_j) with $x_i^{LP} + x_j^{LP}$ fractional. However, given that there are $\mathcal{O}(n^2)$ potential candidates, the chance that a small subset contains a good pair of variables to branch on is small. Therefore, we adopt the following scheme. Given that the variable x_{i^*} with the highest SB score is highly likely to result in effective branching if used by itself, we include it in all the candidate pairs of variables, i.e., we consider only pairs of the form (x_{i^*}, x_j) with $x_{i^*}^{LP} + x_j^{LP}$ fractional. Preliminary computational experiments revealed that it is also beneficial to consider only variables x_j that are active in the solution to the LP relaxation, i.e. with $x_j^{LP} > 0$. These experiments also showed that there can still be a prohibitively large number of variable pairs satisfying these properties. Hence we limit the number whose SB score is evaluated by choosing at most M variables x_j with $x_j^{LP} > 0$ and $x_{i^*}^{LP} + x_j^{LP}$ fractional, where M is a parameter. Finally, let (x_{i^*}, x_{j^*}) be the variable pair from amongst these with the largest SB score $\hat{s}^*$ and let the SB score of x_{i^*} be s^*. If $\hat{s}^* > s^*$, then we choose to branch on the pair (x_{i^*}, x_{j^*}), otherwise, we choose to branch on x_{i^*}. We refer to this selection scheme as GSB: it is summarized diagrammatically in Figure 2.

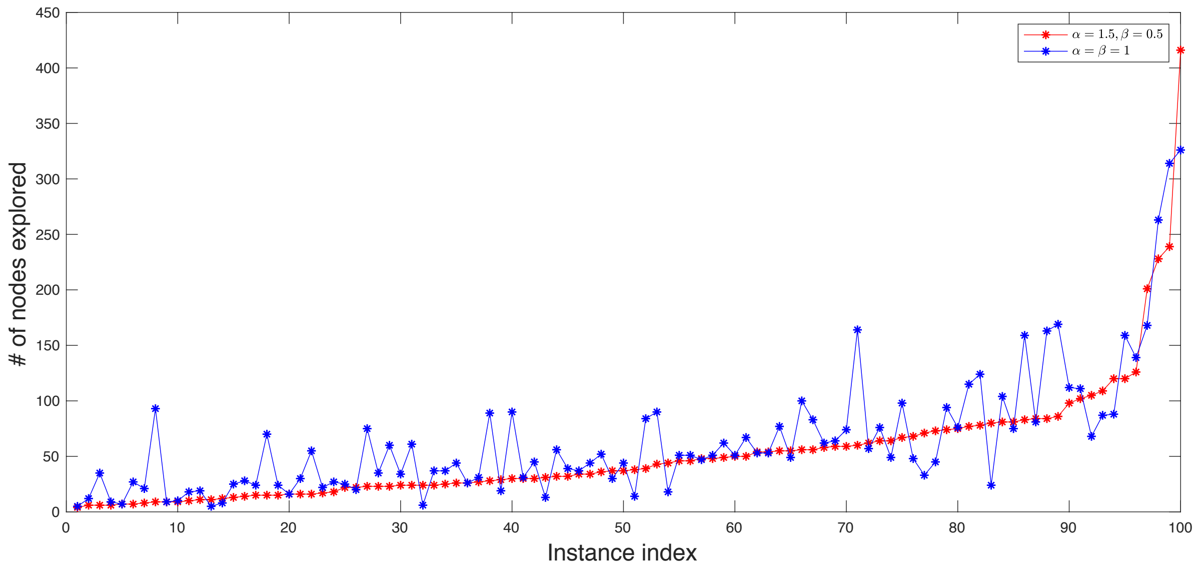


Figure 1: A comparison of the weighted product score function and the simple product score function on 100 randomly generated set covering instances.

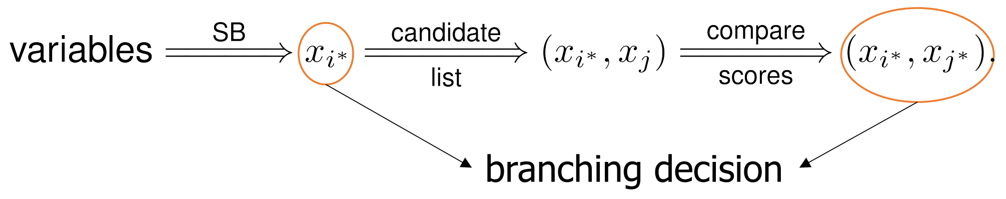


Figure 2: Practical implementation of GSB.

To assess the performance of the GSB selection scheme, where we randomly generate $M = 50$ candidate pairs, we compare its performance with three other selection schemes on 100 randomly generated set covering instances. The other selection schemes proceed as follows. Scheme 1 randomly selects 100 pairs (x_i, x_j) with $x_i^{LP} + x_j^{LP}$ fractional, which implies that at least one of the variables is active in the LP relaxation. Scheme 2 randomly selects 100 pairs (x_i, x_j) with $x_i^{LP} + x_j^{LP}$ fractional and $x_i^{LP} > 0$ and $x_j^{LP} > 0$; both variables are active. Scheme 3 selects randomly selects 100 pairs (x_i, x_j) with $x_i^{LP} + x_j^{LP}$ fractional and $x_i^{LP} + x_j^{LP} > 1$, so both variables are active and at least one has a "high" value. These schemes are given in order of increasing restrictions on the variable pairs considered, which correspond to increasing "activity" of the variables in the LP relaxation. As for GSB, the SB score of every pair selected is calculated, and the pair with highest score is used for branching, unless its score does not exceed that of the best single-variable SB score, in which case the single variable is branched on.

	GSB	Scheme 1	Scheme 2	Scheme 3
Total # of nodes explored	5286	9687	6216	5570
# of times with smallest search tree	51	6	19	24

Table 1: Comparison of candidate pair selection schemes.

As variables selected are increasingly restricted to those that are "more active" in the LP relaxation, the performance of the method improves, with decreasing number of nodes explored. However, even with only half the potential pairs to consider (at most 50 versus 100), the GSB selection scheme explores fewer nodes and most often results in the smallest search tree than the best of the other schemes. This confirms that always including the variable with the highest SB score is beneficial.

4 Learning Generalized Strong Branching

To decide the candidate sets in GSB, we first identify the variable with the highest SB score. Therefore, learning strong branching is a prerequisite to learning generalize strong branching.

4.1 Learning Strong Branching

4.1.1 Methodology

We train a separate model that mimics SB for each problem class and the learning is performed offline. More specifically, we start by using our implementation of SB to solve instances in a training set. At the current node of the search tree, our implementation of SB selects $\ell = \min\{50, v\}$ integer variables with a fractional value in the solution to the LP relaxation, where v is the number of such variables[1], computes their SB scores, and then selects the one with the largest SB score to branch on. The computation of each SB score requires the solution of two LP relaxations. For each variable x_i in the candidate set, we collect a feature vector $\mathbf{f}_i$ and its SB score s_i. Since the ranking by SB scores suffices to select the variable to branch on, predicting the actual SB scores is not necessary. Predicting the exact ranking is not necessary either, because candidates with similar SB scores are expected to behave similarly, and, thus, can be ranked similarly. Doing so introduces some flexibility and tolerance for errors in the model.

We transform a set of SB scores $\{s_{i_1}, \ldots, s_{i_\ell}\}$ into an r-level ranking by a mapping

$$g : \{(1, s_{i_1}), \ldots, (\ell, s_{i_\ell})\} \to \{1, \ldots, r\}.$$

Without loss of generality, we suppose $s_{i_1} \geq s_{i_2} \geq \ldots \geq s_{i_\ell}$. Since we are particularly interested in the candidates with the highest SB scores, we assign rank 1 to the candidates with SB score greater than ηs_{i_1}, where $\eta \in (0, 1)$ and close to 1, a strategy also adopted by Khalil et al. (2016). Since SB score is not a perfect predictor of the benefit of branching on a variable, it is (too) risky to assign rank 1 to just one or two variables, which can happen if there is only one score that is significantly larger than others. To avoid overfitting and thus improve robustness in the training process, we assign rank 1 to at least the top t variables. More specifically, if $k' = \arg\max_k\{s_{i_k} > \eta s_{i_1}\}$ and $k' < t$, then we assign rank 1 to all t candidates with highest SB scores. Thus we assign rank 1 to $t' = \max\{t, k'\}$ variables. We go further

[1]If $v > 50$, the 50 variables are selected at random from among the set of v variables, with each having equal probability of selection.

than the binary labeling scheme in Khalil et al. (2016). The remaining candidates are ranked from 2 to r by evenly distributed quantile:

$$g(k, s_{i_k}) = \begin{cases} 1, & 1 \le k \le t' \\ p+1, & t'+1 \le k \le \ell, \quad s_{i_{t'}} - pq < s_{i_k} \le s_{i_{t'}} - (p-1)q, \end{cases}$$

where $q = \frac{s_{i_{t'}} - s_{i_\ell}}{r-1} + \epsilon$, $p \in \{1, 2, \ldots, r-1\}$, $\epsilon = 10^{-15}$.

Our goal is to train a model that takes in features $(\mathbf{f}_{i_1}, \ldots, \mathbf{f}_{i_\ell})$ of the ℓ candidate variables and predicts a ranking $(r_{i_1}, \ldots, r_{i_\ell})$ close to the true ranking g. We use extreme gradient boosting (XGBoost), an optimized distributed gradient boosting library (see Chen and Guestrin 2016), to accomplish this goal. In addition to its high efficacy and efficiency, proved in various ML competitions[2], XGBoost offers an API in the C language. This is essential for our numerical experiments, since we work with the CPLEX C API to take advantage of its built-in callback functions. In our training, we choose the learning objective to be *rank:pairwise*, which uses *LambdaMART* (Wu et al. 2010) to obtain a ranking. We also tested the other two objective functions available in XGBoost, i.e., *rank:ndcg* and *rank:map*, but they yielded worse performance in terms of the XGBoost performance metric (ndcg) and the branching scheme performance metrics (runtime and search tree size).

In the training process, we solve a set of instances (the training set, TS) to obtain the data for training a ML model. The set TS is split into two subsets: a set, denoted by ML-train, on which the ML algorithm is trained and a set, denoted ML-val, on which the training quality is evaluated by ndcg. We use another, distinct, set of instances (the validation set, VS) to assess the performance of the solver when using the learned model to make branching decisions.

To achieve the best performance on the set VS, we perform a two-stage tuning process. The first stage uses the set ML-val and the metric is ndcg, while the second uses the set VS and the metrics are the runtime and the number of nodes explored, which are what we really care about. We tune the following hyperparameters of XGBoost: *max_depth* for the maximum depth of a tree, *eta* for learning rate, *tree_method* for the tree construction algorithm used in XGBoost.

After tuning has been completed, the performance of the learned model is finally evaluated on a set of instances (the testing set TEST) that have not been used in any part of the training process.

4.1.2 Features

As computational efficiency is critical, and since features can be expensive to compute, we start from a set of 25 features that we believe can provide meaningful information to predict the ranking. The 25 features can be grouped into three categories: static features, dynamic "history-free" features, and dynamic "history-dependent" features. Static features do not change during the solution process and can be computed once before the solution process starts. The dynamic features change during the solution process and need to be computed or updated at every node of the search tree that is not fathomed. Dynamic "history-free" features only use information at the current node, whereas dynamic "history-dependent" features embed information from previously evaluated nodes in the search tree. History-dependent features, especially, require more computational effort. Hence, even if they have high predictive value, their use may not necessarily result in more efficient branching schemes.

Several of these 25 features have been employed in earlier work on the use of ML concepts to develop effective and efficient branching strategies, i.e., Àlvarez et al. (2014), Khalil et al. (2016), Àlvarez et al.

[2]https://github.com/dmlc/xgboost/tree/master/demo#machine-learning-challenge-winning-solutions

(2017), and Gasse et al. (2019), but some are used here for the first time. Furthermore, even if a feature has been employed in earlier research, it may not have been used in exactly the same form; details on normalization, for example, are often not provided. Distinguishing three categories of features is also common, but the terms used to describe the categories are sometimes different. For example, what we refer to a as dynamic history-dependent features are referred to in Àlvarez et al. (2014) as dynamic optimization features.

In defining the 25 features, we ensure that all features are independent of instance size and parameter scale by normalizing feature values with the average of their values over all candidate variables. We use the following notation. Let z_0, z, z_j^- and z_j^+ be the objective values of the LP relaxations at the root node, at the current node, at the down branch, and at the up branch, if variable x_j is branched on, respectively. Furthermore, let c_j be the coefficient of variable x_j in the objective function, A_{ij} be the coefficient of variable x_j in the i-th constraint, and x^{LP} be the solution to the LP relaxation at the current node. Finally, let m and n be the number of constraints and variables, respectively, and $[m] = \{1, \dots, m\}$ and $[n] = \{1, \dots, n\}$. The 25 features associated with variable x_j are as follows.

Static features

- f_1: Number of constraints x_j participates in divided by the total number of constraints.
- f_2: $\frac{c_j - \min_{k\in[n]} c_k}{\max_{k\in[n]} c_k - \min_{k\in[n]} c_k}$.
- f_3: $\frac{f_2}{\text{mean}(\{A_{ij},\ A_{ij}\neq 0,\ i\in[m]\})}$.
- f_4, f_5, f_6: Mean, min, max of $\frac{A_{ij}}{\sum_{k\in[n]} A_{ik}/n}$ over $i \in [m]$.

Dynamic "history-free" features

Node-based

- f_7: Depth of the current node.
- f_8: Current node gap, i.e., $|z - z_0|/|z_0|$.
- f_9: Current node infeasibility, i.e., $\sum_{k\in[n]} \min\{x_k^{LP} - \lfloor x_k^{LP} \rfloor, \lceil x_k^{LP} \rceil - x_k^{LP}\}$ divided by the number of fractional variables in the solution to the LP relaxation.

Variable-based

- f_{10}: $x_j^{LP} - \lfloor x_j^{LP} \rfloor$.
- f_{11}: $\lceil x_j^{LP} \rceil - x_j^{LP}$.
- f_{12}: $x_j^{LP} \cdot f_2$.
- f_{13}, f_{14}, f_{15}: Pseudo costs of x_j (up, down, geometric mean) divided by z.
- f_{16}: Number of times x_j appears in a constraint that is binding divided by the number of binding constraints.

Dynamic "history-dependent" features

Variable-based

- f_{17}: Number of times x_j has been branched on divided by the number of nodes that required branching (so far).
- f_{18}: Number of times x_j appears in the basis divided by number of nodes evaluated (so far).

- f_{19}, f_{20}: Average improvement of the bound on the up and down branches when branching on x_j, i.e., of $|z_j^+ - z|/z$ and $|z_j^- - z|/z$ (so far).
- f_{21}: Average of the reduced cost of x_j divided by z (so far).
- f_{22}, f_{23}, f_{24}: Average pseudo costs (down, up, geometric mean) of x_j divided by z (so far).
- f_{25}: Average of the value of x_j in the solution to the LP relaxation (so far).

As mentioned above, several of these 25 features are used in one or more of Àlvarez et al. (2014), Khalil et al. (2016), Àlvarez et al. (2017) and Gasse et al. (2019), although not necessarily in the same form (in particular, normalizations may differ). Other features are introduced here, based on our knowledge of the domain. For example, we introduce three node-based features, f_7, f_8 and f_9, related to depth, gap, and infeasibility of the LP solution, because past work has shown these criteria have been useful in branch-and-bound search strategies (Linderoth and Savelsbergh 1999, Achterberg et al. 2005, Achterberg 2007). As far as we are aware, features f_1, f_7, f_8, f_9, f_{12}, f_{16} and f_{18} are all new; they have not appeared in previous papers. Furthermore, we believe this is the first time history-dependent versions of reduced cost, pseudocost and LP solution values have been proposed, so, in that sense, $f_{21}, \cdots, f_{25}$ are also new here.

4.1.3 Feature Selection

Some features, especially the dynamic "history-dependent" ones, are time-consuming to compute and may not be necessary for training the model. Thus, we perform feature selection to identify features important for the prediction. We use Recursive Feature Elimination (Guyon et al. 2002) to obtain an effective, but efficiently computable feature set, relying on the importance score provided by XGBoost.

Once a model is trained, XGBoost can report an importance score for each feature as shown on the left in Figure 3. Our feature selection algorithm uses these importance scores and proceeds as follows. We repeatedly drop features that have the smallest importance score in the current model and train a new model using the remaining features. For each model, we compute the prediction accuracy, defined as the ratio of the correct ranking predictions and the total number of ranking predictions, where a predicted ranking is deemed correct if an item ranked 1 in the prediction is in the top 5 of the true ranking. We seek to identify a model with a high prediction accuracy and a small number of features. More specifically, we select a model with at least five features and more if a significant drop (more than 20%) in accuracy occurs. A plot of the accuracy changes in the feature selection process of learning SB for set covering problems is shown on the right in Figure 3 – the model with five features is selected.

XGBoost offers five importance scores which can yield different results for our feature selection procedure. We observed that the use of different importance scores usually result in models with similar accuracy even when the features selected are not the same. This suggests that some of the features capture similar information and using different importance scores results in different sets of features that jointly capture the necessary information. Since the resulting accuracy is not sensitive to the type of importance score in use, we decide to use the default importance type 'weight', which is the number of times a feature is used to split the data across all trees, for our feature selection.

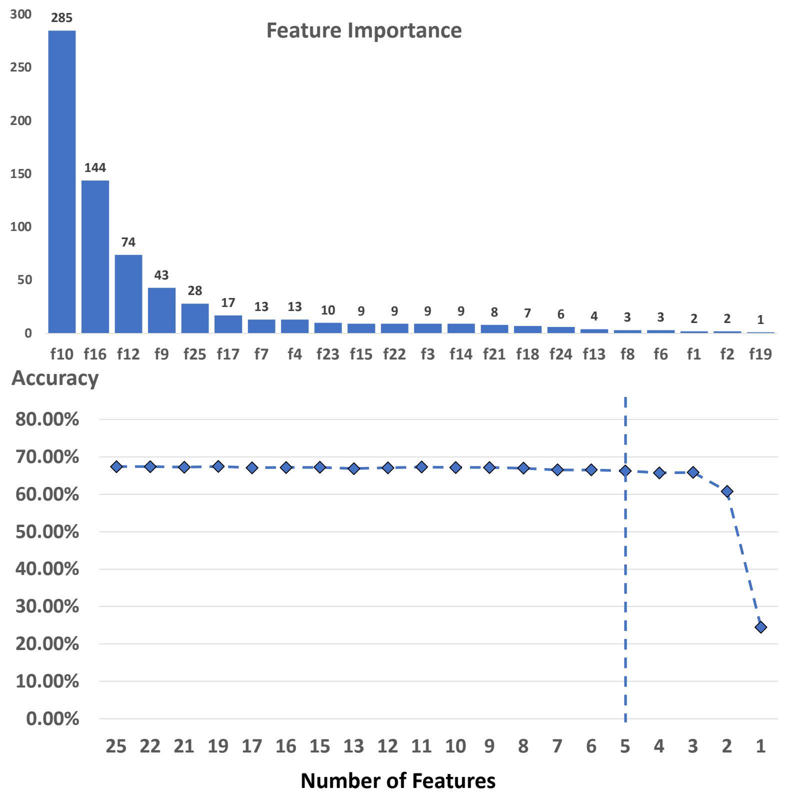


Figure 3: Feature selection of learning SB for set covering problems. At the top: importance scores of the original 25 features in the first model trained (features with importance score 0 are not shown). At the bottom: accuracy changes in the feature selection process. Note that more than one feature may be dropped each time depending on the importance score.

4.2 Learning Generalized Strong Branching

4.2.1 Methodology

Similar to learning SB, we train a model that mimics GSB for each problem class and the learning is performed offline. We start by using GSB (as described in Section 3) to solve instances in a training set. For each candidate pair of variables (x_i, x_j), we collect a feature vector $\hat{\mathbf{f}}_{ij}$ and its SB score $\hat{s}_{ij}$. We introduce an artificial pair of variables (x_{i^*}, x_{i^*}) with SB score s_{i^*}, which represents branching on a single variable (i.e., the variable selected by SB). This captures our desire to branch on the variable x_{i^*} if its SB score s_{i^*} is larger than SB score $\hat{s}_{i^*j^*}$ of the the candidate pair of variables (x_{i^*}, x_{j^*}) with the highest score. Again, we train a model LRN-GSB using XGBoost which takes in features $(\hat{\mathbf{f}}_{i^*j_1}, \ldots, \hat{\mathbf{f}}_{i^*j_\ell}, \hat{\mathbf{f}}_{i^*i^*})$ and predicts a ranking $(r_{j_1}, \ldots, r_{j_\ell}, r_{i^*})$. If $r_{i^*} = 1$, then we branch on the variable x_{i^*}, otherwise, we branch on a variable pair (x_{i^*}, x_j) with $r_j = 1$.

4.2.2 Features

Let u be the number of single-variable features resulting from the feature selection process of learning SB in Section 4.1. For each candidate pair (x_{i^*}, x_j), we collect the following $(6+2u)$ features. The first 6 features are features that seek to capture the interaction of the two variables x_{i^*} and x_j. The next u features are single-variable features of the variable x_j and the last u features are pairwise products of single-variable features of the two variables x_{i^*} and x_j. For the artificial pair (x_{i^*}, x_{i^*}), the first 6 and last u features are set to 0.

Interaction features

- f'_1: $\frac{|c_{i^*}-c_j|}{\max_{k\in[n]} c_k - \min_{k\in[n]} c_k}$.
- f'_2: Number of times x_{i^*} and x_j appear in the same constraint divided by the number of constraints.
- f'_3: Number of times x_{i^*} and x_j appear in the same binding constraint divided by the number of binding constraints.
- f'_4: $x_{i^*}^{LP} + x_j^{LP} - \lfloor x_{i^*}^{LP} + x_j^{LP} \rfloor$.
- f'_5: Indicator function $\mathbb{1}_{\{x_{i^*}^{LP}+x_j^{LP}>1\}}$.
- f'_6: Indicator function $\mathbb{1}_{\{i=j\}}$.

Single-variable features

- $f'_7, \dots, f'_{6+u}$: The u features of the second variable x_j.

Transformed single-variable features

- $f'_{7+u}, \dots, f'_{6+2u}$: Pairwise product of the u features of the two variables x_{i^*} and x_j.

4.2.3 Feature selection

We go through the same feature selection process as described in Section 4.1.3.

5 Computational Study

5.1 Settings

We use CPLEX 12.8 for all experiments. We selected CPLEX from among the many choices of solver platform available since it is representative of the leading group of solvers. This group includes FICO Xpress and Gurobi. Both these solvers and CPLEX outperform other solvers by some margin in independent benchmarking tests (see, for example, Mittelmann (2018), available at `http://plato.asu.edu/talks/informs2018.pdf`). They show relatively similar performance on benchmark instances and it is recognized as extremely difficult to outperform default CPLEX in computational tests.

To be able to focus on the impact of the branching strategy, we allow cuts to be added only at the root and turn off heuristics. We also turn off root presolve as variable aggregation can significantly alter the structure of an instance. All performance evaluation experiments have a node limit of 10^6 and are conducted using a single thread to ensure a fair comparison. We observed in our experiments that both LRN-SB and LRN-GSB are able to nd an optimal solution for all instances tested before reaching the node limit and, thus, there is no reason to include the solution quality in the reported results. We use $\alpha = 1.5$ and $\beta = 0.5$ in the score function (1).

Let CPLEX-D denote CPLEX with default branching, let CPLEX-SB denote CPLEX with its implementation of SB, let OUR-SB denote CPLEX with our implementation of SB, and let LRN-SB and LRN-GSB denote CPLEX with default branching overwritten by the learned SB and learned GSB-2, respectively.

5.2 Set Covering Problem

We consider the set covering problem, i.e., integer programs of the following form:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \geq \vec{\mathbf{1}}, \\ & x \in \{0,1\}^n, \end{aligned}$$

where $c \in \mathbb{R}^n$, A is a $m \times n$ (0,1)-matrix. When we branch on the variable pair (x_i, x_j) and $x_i^{LP} + x_j^{LP} \in (0,1)$, then the down branch will have more freedom than the up branch, and the opposite is true when $x_i^{LP} + x_j^{LP} \in (1,2)$. This observation is used when evaluating the score function (1).

5.2.1 Instances

We have two classes of instances, OR-Library instances and randomly generated instances. All randomly generated instances have $n = 10m$. The cost c_j of variable x_j for $j = 1, \dots, n$ is drawn from a discrete uniform distribution on $\{1, 2, \dots, 100\}$. The number of nonzero elements n_i in row i of A is drawn from a discrete uniform distribution on $\{\frac{2n}{25} + 1, \dots, \frac{3n}{25} - 1\}$, and coefficient a_{ij} is set to 1 with probability $\frac{n_i}{n}$. Given that $n = 10m$, we have that the density of A is about 10%.

The details of the instances used in training, validation, and testing are shown in Table 2, where w indicates the density of the coefficient matrix and q the number of instances in the subset.

5.2.2 Training and feature selection

We use strategy SB to solve instances in training set TS1 to generate data to train LRN-SB. The feature selection process described in Section 4.1.3 results in the following five features being chosen:

1. Current node infeasibility, i.e., $\sum_{k \in [n]} \min\{x_k^{LP} - \lfloor x_k^{LP} \rfloor, \lceil x_k^{LP} \rceil - x_k^{LP}\}$ divided by the number of fractional variables in the solution to the LP relaxation (f_9).
2. $x_j^{LP} - \lfloor x_j^{LP} \rfloor$ (f_{10}).
3. $\frac{x_j^{LP} \cdot (c_j - \min_{k \in [n]} c_k)}{\max_{k \in [n]} c_k - \min_{k \in [n]} c_k}$ (f_{12}).
4. Number of times x_j appears in a constraint that is binding divided by the number of binding constraints (f_{16}).
5. Average of the value of x_j in the solution to the LP relaxation so far (f_{25}).

TS1	m	n	w	q
Random_a1	300	3000	10%	20
Random_a2	400	4000	10%	20
OR-LIB_a1	200	1000	5%	5
OR-LIB_a2	200	2000	2%	5
OR-LIB_a3	300	3000	5%	5
OR-LIB_a4	300	3000	2%	5

TS2	m	n	w	q
Random_a1	300	3000	10%	20

VS	m	n	w	q
Random_b1	400	4000	10%	10
Random_b2	500	5000	10%	10
Random_b3	600	6000	10%	10

TEST	m	n	w	q
Random_c1	300	3000	10%	20
Random_c2	400	4000	10%	20
Random_c3	500	5000	10%	20
Random_c4	600	6000	10%	20
OR-LIB_b1	500	5000	10%	5
OR-LIB_b2	500	5000	20%	5

Table 2: Details of the instances used in training, validation, and testing.

Next, we use strategy GSB to solve instances in training set TS2, with TS2 a small subset of TS1, to generate to train LRN-GSB. The reason why we use a small subset is that solving instances using strategy GSB is very time-consuming. Feature selection results in the following five features being chosen (Figure 4 presents information related to the feature selection process):

1. $x_{i^*}^{LP} + x_j^{LP} - \lfloor x_{i^*}^{LP} + x_j^{LP} \rfloor$ (f'_4).
2. $x_j^{LP} - \lfloor x_j^{LP} \rfloor$ (f'_8).
3. Number of times x_j appears in a constraint that is binding divided by the number of binding constraints (f'_{10}).
4. $(x_{i^*}^{LP} - \lfloor x_{i^*}^{LP} \rfloor) \cdot (x_j^{LP} - \lfloor x_j^{LP} \rfloor)$ (f'_{13}).
5. Average of the value of x_j in the solution to the LP relaxation so far times average of the value of x_{i^*} in the solution to the LP relaxation so far (f'_{16}).

Instances in the validation set VS are used to tune XGBoost to train LRN-SB and LRN-GSB for better performance. Finally, the performance of trained models is evaluated by solving instances in the test set TEST. For randomly generated instances, the instance used for validation and testing are different.

5.2.3 Results

To measure the accuracy of the learned models, we use "Top k accuracy%" defined as the ratio of number of times LRN-SB branches on a variable in the top k of the ranking computed by OUR-SB and the total number of nodes in the search tree that were not fathomed.

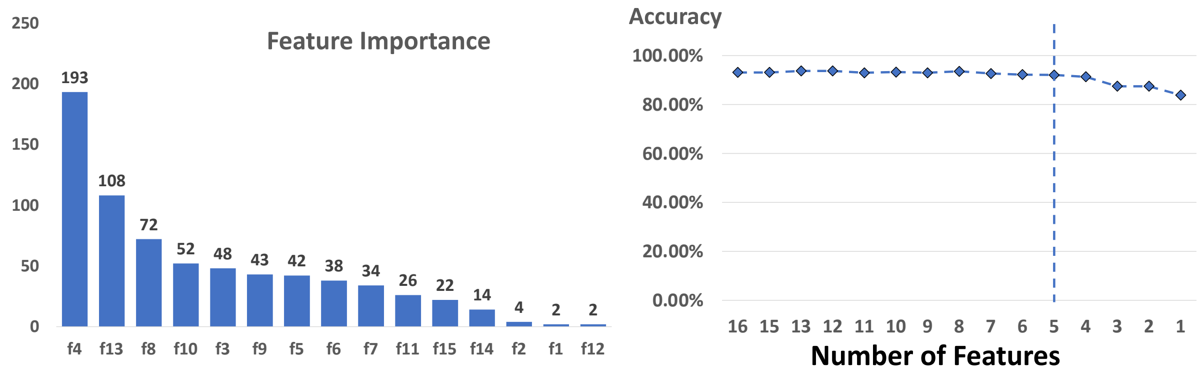


Figure 4: Feature selection of learning GSB for set covering problems. On the left: importance scores of the original 16 features in the first model trained (features with importance score 0 are not shown). On the right: accuracy changes in the feature selection process.

The results for the instances in the test set can be found in Table 3 and 4. We report totals for each of the subsets, and, for convenience, also the percentage reduction compared to CPLEX with default branching – the largest percentage reduction is highlighted using the underline style.

Name	Number of nodes explored					Top 5
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB	Accuracy
Random_c1	17,005	3,415	7,862	10,120	9,437	65.73%
Savings	—	79.92%	53.77%	40.49%	44.50%	
Random_c2	83,630	20,673	38,808	50,763	50,087	67.21%
Savings	—	75.28%	53.60%	39.30%	40.11%	
Random_c3	677,597	147,493	289,853	363,609	294,524	66.71%
Savings	—	78.23%	57.22%	46.34%	56.53%	
Random_c4	6,042,591	1,441,345	4,121,957	3,005,899	2,660,864	66.59%
Savings	—	76.15%	31.78%	50.25%	55.96%	
OR-LIB_b1	379,338	91,224	147,818	248,353	180,839	67.24%
Savings	—	75.95%	61.03%	34.53%	52.33%	
OR-LIB_b2	502,572	49,353	284,273	86,598	190,993	74.40%
Savings	—	90.18%	43.44%	82.77%	62.00%	
Total	7,702,733	1,753,503	4,890,571	3,765,342	3,386,744	66.95%
Savings	—	<u>77.24%</u>	36.51%	51.12%	56.03%	

Table 3: Comparison of the number of nodes explored for set covering instances.

We observe that LRN-SB significantly outperforms CPLEX-D, on average a reduction of around 51% in terms of the number of nodes explored and of almost 39% in terms of computing time. LRN-GSB performs even better in terms of the number of nodes explored and on average, the reduction is around 56%. As for the computing times, the reduction is around 37%. In Figure 5, we provide more detail and report the performance of CPLEX-D and LRN-GSB for all 90 test instances. We see that LRN-GSB consistently performs better than CPLEX-D (the values are on a logarithm scale, so the the actual differences are much larger than what is shown in the figure).

To further assess the performance of LRN-GSB, we generate and solve additional large instances (with $m = 600$ and $n = 6000$), but with different matrix densities (with $w = 5\%$ and $w = 20\%$). The

	Run time (seconds)				
Name	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB
Random_c1	24.01	65.95	222.53	16.44	16.52
Savings	—	-174.68%	-826.82%	31.53%	31.20%
Random_c2	117.77	497.69	1,225.79	86.61	94.36
savings	—	-322.59%	-940.83%	26.46%	19.88%
Random_c3	1,087.91	4,248.86	11,916.94	738.8	675.07
Savings	—	-290.55%	-995.40%	32.09%	37.95%
Random_c4	12,608.06	57,573.18	231,886.18	7,747.28	7,940.78
Savings	—	-356.64%	-1739.19%	38.55%	37.02%
OR-LIB_b1	566.12	2,648.44	6,233.16	491.19	390.28
Savings	—	-367.82%	-1001.03%	13.24%	31.06%
OR-LIB_b2	659.74	1,642.62	25,724.2	173.53	388.76
Savings	—	-148.98%	-3799.14%	73.70%	41.07%
Total	15,063.61	66,676.74	277,208.8	9,253.85	9,505.77
Savings	—	-342.63%	-1740.25%	38.57%	36.90%

Table 4: Comparison of the solution times for set covering instances.

results can be found in Table 5, where, for completeness sake, we also report the results for matrix density $w = 10\%$. We observe that for all matrix density levels, LRN-GSB consistently and significantly outperforms CPLEX-D. As CPLEX-D reaches the limit of one million nodes in the search tree for 3 of the 20 instances with $w = 20\%$, the results are even somewhat biased in favor of CPLEX-D for $w = 20\%$. What is especially interesting about the results for $w = 20\%$ is that none of the instances used in training were this large and had such high matrix density; however, the learned model still performs extremely well.

	Number of nodes explored		Run time (seconds)	
w	CPLEX-D	CPLEX-GSB	CPLEX-D	CPLEX-GSB
5%	1,307,090	550,800	2,809.15	1,667.00
Savings	—	57.86%	—	40.66%
10%	6,042,591	2,660,864	12,608.06	7,940.78
Savings	—	55.96%	—	37.02%
20%	8,793,859	2779239	17107.46	7780.01
Savings	—	68.40%	—	56.05%

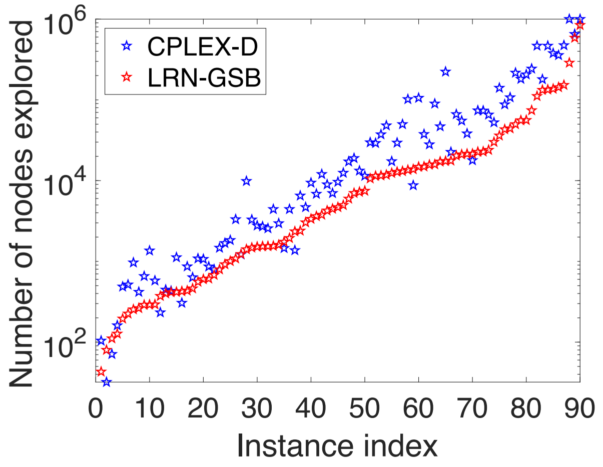
Table 5: Comparison of number of nodes explored and solution times for large set covering instances with $m = 600$, $n = 6000$.

5.3 Set Packing Problem

Next, we consider the set packing problem, i.e., integer programs of the form:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq \vec{\mathbf{1}}, \\ & x \in \{0,1\}^n, \end{aligned}$$

where $c \in \mathbb{R}^n$, A is a $m \times n$ (0,1)-matrix. Consider a constraint $\sum_{k \in K} x_k \leq 1$. When we branch on the variable pair (x_i, x_j) with $i, j \in K$, on the down branch x_i and x_j are fixed to 0, but on the up



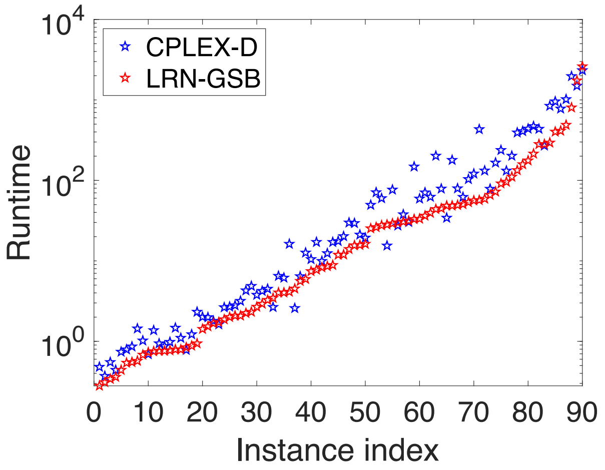


Figure 5: Performance of CPLEX-D and LRN-GSB on all instances in the test set (sorted in order of non-decreasing LRN-GSB values). On the left: Number of nodes explored. On the right: Computing time. The y-axis is set to log scale.

branch all variables x_k with $k \in K \setminus \{i, j\}$ are fixed to 0. Thus, the down branch has more freedom. This observation is used when evaluating the score function (1).

5.3.1 Instances

Instances are randomly generated in the same way as in 5.2.1 except that $n = 5m$. The details of the instances used in training, validation, and testing are shown in Table 6.

TS1	m	n	w	q
Random_d1	100	500	10%	20
Random_d2	150	750	10%	20
Random_d3	200	1000	10%	20

TS2	m	n	w	q
Random_e1	100	500	10%	10
Random_e2	150	750	10%	10

VS	m	n	w	q
Random_f1	150	750	10%	10
Random_f2	200	1000	10%	10
Random_f3	250	1250	10%	10

TEST	m	n	w	q
Random_g1	100	500	10%	20
Random_g2	150	750	10%	20
Random_g3	200	1000	10%	20
Random_g4	250	1250	10%	20
Random_g5	300	1500	10%	20

Table 6: Details of the instances used in training, validation, and testing.

5.3.2 Training and feature selection

Using feature selection as described in Section 4.1.3, the following five features are selected for training LRN-SB:

1. Number of constraints in which x_j appears divided by the number of constraints (f_1).
2. $x_j^{LP} - \lfloor x_j^{LP} \rfloor$ (f_{10}).
3. $\lceil x_j^{LP} \rceil - x_j^{LP}$ (f_{11}).
4. Number of times x_j appears in a constraint that is binding divided by the number of binding constraints (f_{16}).
5. Average of the value of x_j in the solution to the LP relaxation (so far) (f_{25}).

Similarly, the following five features are selected for training LRN-GSB.

1. Number of times x_{i^*} and x_j appear in the same constraint divided by the number of constraints (f'_2).
2. $x_{i^*}^{LP} + x_j^{LP} - \lfloor x_{i^*}^{LP} + x_j^{LP} \rfloor$ (f'_4).
3. $x_j^{LP} - \lfloor x_j^{LP} \rfloor$ (f'_8).
4. $(x_{i^*}^{LP} - \lfloor x_{i^*}^{LP} \rfloor) \cdot (x_j^{LP} - \lfloor x_j^{LP} \rfloor)$ (f'_{13}).
5. $(\lceil x_{i^*}^{LP} \rceil - x_{i^*}^{LP}) \cdot (\lceil x_j^{LP} \rceil - x_j^{LP})$ (f'_{14}).

5.3.3 Results

The results on the instances in the test set can be found in Table 7 and Table 8.

Name	Number of nodes explored					Top 5
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB	Accuracy
Random1_g1	35,489	5,604	13,348	18,008	21,659	57.24%
savings	—	84.21%	62.39%	49.26%	38.97%	
Random1_g2	187,742	44,354	79,289	105,175	100,326	67.53%
savings	—	76.38%	57.77%	43.98%	46.56%	
Random1_g3	217,579	72,191	108,008	134,939	124,652	77.43%
savings	—	66.82%	50.36%	37.98%	42.71%	
Random1_g4	344,339	103,481	165,997	197,382	203,088	81.77%
savings	—	69.95%	51.79%	42.68%	41.02%	
Random1_g5	345,463	104,276	168,075	184,140	183,097	83.91%
savings	—	69.82%	51.35%	46.70%	47.00%	
Total	1,130,612	329,906	534,717	639,644	632,822	73.58%
Savings	—	<u>70.82%</u>	52.71%	43.42%	44.03%	

Table 7: Comparison on the number of nodes explored for set packing instances.

We see that LRN-GSB outperforms CPLEX-D both in terms of the number of nodes explored and the computing time. However, the improvements are not as significant, on average, a 44% reduction in the number of nodes explored and a 12% reduction in computing time. Although LRN-GSB, on average, explores a smaller number of nodes than LRN-SB, its computing time, on average, is slightly higher than

Name	Run time (seconds)				
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB
Random1_g1	24.44	64.6	137.54	14.75	19.26
savings	—	-164.32%	-462.77%	39.65%	21.19%
Random1_g2	217.02	823.88	1,576.46	159.92	166.8
savings	—	-279.63%	-626.41%	26.31%	23.14%
Random1_g3	549.65	1,993.62	4,039.72	434.57	442.44
savings	—	-262.71%	-634.96%	20.94%	19.51%
Random1_g4	991.08	3,714.48	9,016.01	833.85	862.34
savings	—	-274.79%	-809.72%	15.86%	12.99%
Random1_g5	1,682.15	6,104.38	17,382.97	1,511.29	1,548.72
savings	—	-262.89%	-933.38%	10.16%	7.93%
Total	3,464.34	12,700.96	32,152.7	2,954.38	3,039.56
Savings	—	-266.62%	-828.10%	14.72%	12.26%

Table 8: Comparison of solution times for set packing instances.

LRN-SB. This is because more time is spent on each branching decision. Furthermore, possibly more importantly, LRN-GSB adds constraints to enforce branching decisions, whereas LRN-SB only changes variables bounds to enforce branching decisions. Another interesting observation is that the prediction accuracy of LRN-SB for instances of the set packing problem appears to be higher than for instances of the set covering problem, which suggests that LRN-SB already performs very well and it is more difficult for LRN-GSB to achieve further improvements.

5.4 0-1 Knapsack Problems

Next, we consider the 0-1 knapsack problem, i.e., integer programs of the form:

$$\begin{aligned} \max \quad & p^T x \\ \text{s.t.} \quad & w^T x \leq b, \\ & x \in \{0,1\}^n, \end{aligned}$$

where $p, w \in \mathbb{R}^n$. We make a minor change in the way we select candidate pairs of variables. Rather than requiring that $x_j^{LP} > 0$, we require that x_j is not fixed, i.e., its lower and upper bound at the current node are not equal.

5.4.1 Instances

All instances are randomly generated as follows. The profit p_j and the weight w_j for $j = 1, \dots, n$ are drawn from a discrete uniform distribution on $\{1, 2, \dots, 10n\}$ and we set $b = \lfloor \sum_{i=1}^n w_i / 5 \rfloor$. The details of the instances used in training, evaluating, and testing are shown in Table 9.

5.4.2 Training and feature selection

The five features selected for training LRN-SB are as follows:

1. $\frac{c_j - \min_{k \in [n]} c_k}{\max_{k \in [n]} c_k - \min_{k \in [n]} c_k}$ (f_2).
2. Mean of $\frac{A_{ij}}{\sum_{k \in [n]} A_{ik}/n}$ w.r.t $i \in [m]$ (f_4).

TS1	n	q
Random_h1	2000	20
Random_h2	3000	20
Random_h3	4000	20

TS2	n	q
Random_i1	2000	10
Random_i2	3000	10

VS	n	q
Random_j1	2000	10
Random_j2	3000	10

TEST	n	q
Random_k1	2000	20
Random_k2	3000	20
Random_k3	4000	20
Random_k4	5000	20
Random_k5	6000	20

Table 9: Details of the instances used in training, validation, and testing.

3. Max of $\frac{A_{ij}}{\sum_{k\in[n]} A_{ik}/n}$ w.r.t $i \in [m]$ (f_6).
4. Up pseudo-cost of x_j divided by z (f_{13}).
5. Average of value of x_j in the solution to the LP relaxation (so far) (f_{25}).

The five features selected for training LRN-GSB are as follows:

1. Number of times x_{i^*} and x_j appear in the same constraint divided by the number of constraints (f'_2).
2. $x_{i^*}^{LP} + x_j^{LP} - \lfloor x_{i^*}^{LP} + x_j^{LP} \rfloor$ (f'_4).
3. Indicator function $\mathbb{1}_{\{x_{i^*}^{LP}+x_j^{LP}>1\}}$ (f'_5).
4. $\frac{c_j - \min_{k\in[n]} c_k}{\max_{k\in[n]} c_k - \min_{k\in[n]} c_k}$ (f'_7).
5. Product of up pseudo-cost of x_{i^*} and x_j divided by z^2 (f'_{15}).

5.4.3 Results

Even though there is only a single constraint in the 0-1 knapsack problem (which should imply that there is only a single fractional variable in a solution to the LP relaxation), CPLEX-D, CPLEX-SB, OUR-SB, RAND, and LRN-SB can still exhibit different behavior because we allow cuts to be added at the root. However, we have observed that the number of cuts added at the root is very small and so is the number of fractional variables – usually fewer than five. Therefore, we report "Top 1 accuracy%" instead of "Top 5 accuracy%".

The results on the instances in the test set can be found in Table 10 and Table 11. We observe, again, that LRN-SB and LRN-GSB significantly outperform CPLEX-D in terms of both number of nodes explored and computing time, where LRN-GSB is, again, better than LRN-SB. What is most surprising is that CPLEX-SB has the worst performance, even in terms of number of nodes explored.

Name	Number of nodes explored					Top 1 Accuracy
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB	
Random_k1	33,813	121,798	27,219	25,862	21,056	99.62%
Savings	—	-260.21%	19.50%	23.51%	37.73%	
Random_k2	269,145	113,748	37,421	36,372	50,045	99.06%
Savings	—	57.74%	86.10%	86.49%	81.41%	
Random_k3	128,406	153,494	51,736	52,593	55,048	99.39%
savings	—	-19.54%	59.71%	59.04%	57.13%	
Random_k4	401,585	151,681	69,477	65,193	34,219	99.46%
savings	—	62.23%	82.70%	83.77%	91.48%	
Random_k5	403,323	441,979	78,708	86,559	45,403	99.57%
savings	—	-9.58%	80.49%	78.54%	88.74%	
Total	1,236,272	982,700	264,561	266,579	205,771	99.42%
Savings	—	20.51%	78.60%	78.44%	83.36%	

Table 10: Comparison of the number of nodes explored for 0-1 knapsack instances.

Name	Run time (seconds)				
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB
Random_k1	13.46	34.96	46.79	19.96	18.07
Savings	—	-159.73%	-247.62%	-48.29%	-34.25%
Random_k2	88.12	47.39	289.44	41.65	51.11
Savings	—	46.22%	-228.46%	52.73%	42.00%
Random_k3	80.64	72.51	463.88	72.74	72.23
Savings	—	10.08%	-475.25%	9.80%	10.43%
Random_k4	248.98	98.73	698.41	112.33	75.78
Savings	—	60.35%	-180.51%	54.88%	69.56%
Random_k5	301.1	303.76	870.23	172.96	115.32
Savings	—	-0.88%	-189.02%	42.56%	61.70%
Total	732.3	557.35	2,368.75	419.64	332.51
Savings	—	23.89%	-223.47%	42.70%	54.59%

Table 11: Comparison of computing times for 0-1 knapsack instances.

5.5 Observations

The excellent performance of the branching schemes for the three problem classes is enabled by the selected features – those found to be most useful – in the trained models. For learning SB branching, 11 of the 25 features were selected in at least one of the three problem classes: 4 of the 6 static features, 6 of the 10 dynamic history-free features and 1 of the 9 dynamic history-dependent features.

Four features were selected in more than one problem class. The lower and upper fractional parts of a variable in the LP solution (f_{10} and f_{11}), which are widely used, were both selected for set covering and set packing. Another dynamic history-free feature, newly introduced, was also selected for both set covering and set packing: the number of times a variable appears in a binding constraint in the current LP relaxation normalized by the number of such binding constraints (f_{16}). The features selected for 0-1 knapsack problems have little overlap with those selected for set covering and set packing problems (which may not be surprising as constraint coefficients of 0-1 knapsack instances are general integers whereas the constraint coefficients of set covering and set packing instances are either 0 or 1). The exception was the only dynamic history-dependent feature selected: the average value of a variable in all LP relaxation solutions seen so far in the tree (f_{25}), which is one of the new features we introduce here. This feature was selected for both knapsack and set partitioning problems.

In addition to f_{16} and f_{25}, a further 3 of the features newly introduced in this paper were selected: f_1, f_9 and f_{12}. The first of these, f_1, is a static feature giving the proportion of constraints a variable appears in, and was selected for set packing. The only dynamic node-based feature selected, the average infeasibility of a fractional variable in the current LP solution (f_9), was selected for set covering. Another new dynamic history-free feature, which weights the cost of a variable relative to the minimum over all variables (normalized by the cost range) by the value of the variable in the current LP solution (f_{12}) was also selected for set covering.

Interestingly, the features selected in learning GSB-2 for set covering and set partitioning were all either some combination of fractional parts of variables in the current LP relaxation, or were based on one of the new features introduced. Specifically, the GSB-2 versions of f_{12} and f_{16} were selected for set covering and the GSB-2 interaction feature based on f_1 was selected set packing. This feature was also selected for 0-1 knapsack problems, as was another interaction feature that we introduced for GSB-2, f'_5, which indicates whether the sum of two variables in the current LP solution exceeds 1 or not.

We conclude that the new features proposed have played a useful role and expand the set of features worth considering in learned models for branching.

5.6 Default Settings

In the computational experiments present in the previous subsections, we have turned off CPLEX preprocessing, CPLEX heuristics, and CPLEX cut generation in nodes other than the root, in order to be able to “isolate” the effect of the branching strategy. One may argue that provides only limited information on the value of the proposed learned branching strategies. Therefore, we have rerun all set covering and set packing instances, but this time with all default settings of CPLEX, except for the branching scheme. In these experiments, we exclude the 0-1 knapsack instances. CPLEX uses a variety of techniques, e.g., preprocessing, primal heuristics, and cut generation, when solving an instance. Among the cuts that CPLEX generates are lifted 0-1 knapsack covers, which are extremely effective when solving instances of 0-1 knapsack problems. As a result, even solving very large instances of 0-1 knapsack problems only leads to very small search trees, which implies that improved branching schemes have little or no benefit.

5.6.1 Set Covering Problem

The results of the computational experiments can be found in Table 12 and 13. We observe that LRN-SB and LRN-GSB still significantly outperform CPLEX-D, with, on average, a reduction of 52% and 53% in computing time, respectively. In fact, the improvements could be even greater if the learned strategies are fully integrated into CPLEX when its internal data structure is accessible.

	Number of nodes explored					Top 5 Accuracy
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB	
Random_c1	9,238	2,643	5,191	6,272	6,440	65.76%
Savings	—	71.39%	43.81%	32.11%	30.29%	
Random_c2	62,769	17,474	30,884	40,715	40,527	67.07%
savings	—	72.16%	50.80%	35.14%	35.43%	
Random_c3	540,532	126,479	221,347	339,488	297,053	66.76%
Savings	—	76.60%	59.05%	37.19%	45.04%	
Random_c4	5,987,841	1,091,251	1,802,029	2,255,752	2,068,149	66.23%
Savings	—	81.78%	69.91%	62.33%	65.46%	
OR-LIB_b1	473,797	54,144	122,734	210,230	179,626	66.88%
Savings	—	88.57%	74.10%	55.63%	62.09%	
OR-LIB_b2	300,003	67,455	197,604	230,369	132,475	72.86%
Savings	—	77.52%	34.13%	23.21%	55.84%	
Total	7,374,180	1,359,446	2,379,789	3,082,826	2,724,270	66.84%
Savings	—	81.56%	67.73%	58.19%	63.06%	

Table 12: Comparison of the number of nodes explored for set covering instances (default CPLEX settings).

	Run time (seconds)				
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB
Random_c1	24.21	43.32	71.55	19.68	20.35
Savings	—	-78.93%	-195.54%	18.71%	15.94%
Random_c2	96.3	295.66	433.41	78.66	81.95
Savings	—	-207.02%	-350.06%	18.32%	14.90%
Random_c3	772.68	2,611.14	3,846.96	580.91	561.94
Savings	—	-237.93%	-397.87%	24.82%	27.27%
Random_c4	10,841.31	31,735.29	39,472.03	4,747.77	4,768.34
Savings	—	-192.73%	-264.09%	56.21%	56.02%
OR-LIB_b1	636.05	1,162.46	2,270.96	355.49	337.84
Savings	—	-82.76%	-257.04%	44.11%	46.88%
OR-LIB_b2	388.75	1,384.31	3,286.55	334.5	224.93
Savings	—	-256.09%	-745.41%	13.95%	42.14%
Total	12,759.3	37,232.18	49,381.46	6,117.01	5,995.35
Savings	—	-191.80%	-287.02%	52.06%	53.01%

Table 13: Comparison of computing times for set covering instances (default CPLEX settings).

5.6.2 Set Packing Problems

The results of the computational experiments can be found in Table 14 and 15. As with the set covering instances, LRN-SB and LRN-GSB still significantly outperform CPLEX-D, with, on average, a reduction of 26% and 24% in computing time, respectively.

	Number of nodes explored					Top 5
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB	Accuracy
Random_g1	26,189	4,654	11,070	15,937	15,299	57.24%
Savings	—	82.23%	57.73%	39.15%	41.58%	
Random_g2	151,127	39,372	62,337	79,206	78,037	67.53%
Savings	—	73.95%	58.75%	47.59%	48.36%	
Random_g3	213,988	56,722	84,836	94,442	96,379	77.43%
Savings	—	73.49%	60.35%	55.87%	54.96%	
Random_g4	328,638	76,617	138,657	145,188	147,282	81.77%
Savings	—	76.69%	57.81%	55.82%	55.18%	
Random_g5	315,255	77,508	138,175	143,052	142,063	83.91%
Savings	—	75.41%	56.17%	54.62%	54.94%	
Total	1,035,197	254,873	435,075	477,825	479,060	73.58%
Savings	—	75.38%	57.97%	53.84%	53.72%	

Table 14: Comparison of number of nodes explored for set packing instances (default CPLEX settings).

Name	Run time (seconds)				
	CPLEX-D	CPLEX-SB	OUR-SB	LRN-SB	LRN-GSB
Random_g1	28.11	61.58	118.58	20.6	21.77
savings	—	-119.07%	-321.84%	26.72%	22.55%
Random_g2	244.17	778.98	1,322.18	167.79	176.02
savings	—	-219.03%	-441.50%	31.28%	27.91%
Random_g3	674.14	1,879.89	3,731.11	503.74	518.55
savings	—	-178.86%	-453.46%	25.28%	23.08%
Random_g4	1,312.1	3,645.31	9,613.23	985.79	1,004.31
savings	—	-177.82%	-632.66%	24.87%	23.46%
Random_g5	2,284.04	5,989.27	17,901.58	1,704.14	1,737.75
savings	—	-162.22%	-683.77%	25.39%	23.92%
Total	4,542.56	12,355.03	32,686.68	3,382.06	3,458.4
Savings	—	-171.98%	-619.57%	25.55%	23.87%

Table 15: Comparison of computing time for set packing instances (default CPLEX settings).

6 Final Remarks

We have shown that complex branching strategies, specifically, strong branching and generalized strong branching, can be implemented efficiently by incorporating a model that mimics their behavior and is learned offline. The resulting performance, on specific classes of integer programs, is not only competitive with the branching strategies found in a state-of-the-art commercial solver, they outperform these branching strategies significantly (even given the disadvantages of having to implement these branching strategies using callback functions and not having access to customized internal data structures). Equally important is the fact that the learned models are robust in the sense that even though they are trained on only small (easy) instances, the trained models are able to solve large (difficult) instances effectively and efficiently.

Using the ML approach presented in this paper to develop a model that selects branching variable sets of cardinality larger than two becomes too time-consuming. New ideas will be needed to be able to limit the number of variable sets considered, and to do the learning more efficiently. As branching on larger sets of variables has the potential to reduce the size of the search tree even more, this is an interesting direction for future research.

References

T. Achterberg. *Constraint integer programming.* PhD thesis, Technische Universität Berlin, 2007.

T. Achterberg and R. Wunderling. Mixed integer programming: Analyzing 12 years of progress. In *Facets of combinatorial optimization*, pages 449–481. Springer, 2013.

T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1): 42–54, 2005.

A. M. Àlvarez, Q. Louveaux, and L. Wehenkel. A supervised machine learning approach to variable branching in branch-and-bound. Technical report, Université de Liège, 2014.

A. M. Àlvarez, L. Wehenkel, and Q. Louveaux. Online learning for strong branching approximation in branch-and-bound. Technical report, Université de Liège, 2016.

A. M. Àlvarez, Q. Louveaux, and L. Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding cuts in the TSP (a preliminary report). Technical report, Citeseer, 1995.

J. A. Appleget and R. K. Wood. Explicit-constraint branching for solving mixed-integer programs. In *Computing Tools for Modeling, Optimization and Simulation*, pages 245–261. Springer, 2000.

E. Beale. Branch and bound methods for mathematical programming systems. In P. Hammer, E. Johnson, and B. Korte, editors, *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 201 – 219. Elsevier, 1979.

E. M. L. Beale and J. A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In J. Lawrence, editor, *Proceedings 5th IFORS Conference, Tavistock*, pages 447–454. Wiley, 1970.

Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour dhorizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

M. Bénichou, J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971.

D. Bertsimas and B. Stellato. The voice of optimization. *Machine Learning*, 110(2):249–277, 2021.

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

A. Chmiela, E. B. Khalil, A. Gleixner, A. Lodi, and S. Pokutta. Learning to schedule heuristics in branch-and-bound. *arXiv preprint arXiv:2103.10294*, 2021.

M. Fischetti and M. Monaci. Backdoor branching. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 183–191. Springer, 2011.

C. Gambella, B. Ghaddar, and J. Naoum-Sawaya. Optimization problems for machine learning: A survey. *European Journal of Operational Research*, 290(3):807–828, 2021.

M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint arXiv:1906.01629*, 2019.

J.-M. Gauthier and G. Ribière. Experiments in mixed-integer linear programming using pseudo-costs. *Mathematical Programming*, 12(1):26–47, 1977.

I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Machine Learning*, 46:389–422, 2002.

H. He, H. Daume III, and J. M. Eisner. Learning to search in branch and bound algorithms. *Advances in neural information processing systems*, 27:3293–3301, 2014.

Z. Huang, K. Wang, F. Liu, H.-l. Zhen, W. Zhang, M. Yuan, J. Hao, Y. Yu, and J. Wang. Learning to select cuts for efficient mixed-integer programming. *arXiv preprint arXiv:2105.13645*, 2021.

A. Jiménez-Cordero, J. M. Morales, and S. Pineda. Offline constraint screening for online mixed-integer optimization. *arXiv preprint arXiv:2103.13074*, 2021.

M. Karimi-Mamaghan, M. Mohammadi, P. Meyer, A. M. Karimi-Mamaghan, and E.-G. Talbi. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research*, 2021.

E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. Learning to run heuristics in tree search. In *IJCAI*, pages 659–666, 2017.

J. T. Linderoth and M. W. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

A. Lodi and G. Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.

H. Mittelmann. Latest benchmark results. In *INFORMS Annual Conference*, 2018. URL `http://plato.asu.edu/talks/informs2018.pdf`.

G. Pataki and M. Tural. Basis reduction methods. *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

D. M. Ryan and B. A. Foster. An integer programming approach to scheduling. *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, pages 269–280, 1981.

Y. Tang, S. Agrawal, and Y. Faenza. Reinforcement learning for integer programming: Learning to cut. In *International Conference on Machine Learning*, pages 9367–9376. PMLR, 2020.

Q. Wu, C. J. Burges, K. M. Svore, and J. Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 13(3):254–270, 2010.

Y. Yang, N. Boland, and M. Savelsbergh. Multivariable branching: A 0-1 knapsack problem case study. *INFORMS Journal on Computing*, 2021.

K. Yilmaz and N. Yorke-Smith. A study of learning search approximation in mixed integer branch and bound: Node selection in scip. *AI*, 2(2):150–178, 2021.