Online Learning for Strong Branching Approximation in Branch-and-Bound

Alejandro Marcos Alvarez, Louis Wehenkel, and Quentin Louveaux *

Department of EE&CS, Université de Liège, Liège, Belgium
{amarcos,l.wehenkel,q.louveaux}@ulg.ac.be

Abstract. We present an online learning approach to variable branching in branch-and-bound for mixed-integer linear problems. Our approach consists in learning strong branching scores in an online fashion and in using them to take branching decisions. More specifically, numerical scores are used to rank the branching candidates. If, for a given variable, the learned approximation is deemed reliable, then the score for that variable is computed thanks to the learned function. If the approximation is not reliable yet, the real strong branching score is used instead. The scores that are computed through the real strong branching procedure are fed to the online learning algorithm in order to improve the approximated function. The experiments show promising results.

Keywords: branch-and-bound, variable branching, online learning

1 Introduction

Branch-and-bound (B&B) [10] is probably the most widely used algorithm to solve Mixed-Integer Programming (MIP) problems. Despite the conceptual simplicity of B&B, there exist many additional features, such as cutting planes, presolve, heuristics or advanced branching strategies, that can be used to enhance the performance of the algorithm [1]. Among these, the key element that most conditions the efficiency of the method is probably the branching strategy [1].

Because of the huge impact that branching strategies have on the performance of the solvers, this area naturally attracted a lot of attention from the community. The very first branching strategy, known as *most-infeasible branching* [10], consists in choosing as branching variable the variable whose fractional part is closest to 0.5. Later on, more evolved techniques were developed. For instance, *pseudocost branching* [5] is a procedure that estimates the dual bound improvement for a candidate variable based on the dual bound improvements observed during previous branchings. Despite the cleverness of the idea, pseudocost branching is known to make poor choices when not enough historical data

* This work was partially funded by the Dysco IUAP network of the Belgian Science Policy Office and the Pascal2 network of excellence of the EC. The scientific responsibility rests with the authors.

is available. In order to alleviate this problem, Achterberg et al. [2] proposed to explicitly compute the dual bound improvements for those variables that are deemed 'unreliable'. Using this trick to initialize pseudocost branching gave birth to the famous *reliability branching*, which is probably one of the most effective branching strategies ever developed. Finally, let us mention *strong branching* [4] that chooses a branching variable by explicitly computing the dual bound improvements resulting from the candidate branching. Strong branching is known to be very time consuming but traditionally yields very small B&B trees.

Following the footsteps of pseudocost branching, some researchers recently started developing branching procedures relying on information gathered during an exploratory phase. For instance, *backdoor branching* [8] and *information-based branching* [9] collect information in a first phase consisting in multiple restarts of the optimization procedure for a given problem. During this first phase, some information is harvested and is then used in a second phase for the 'real' optimization of the problem. In another line of research, Marcos Alvarez et al. [11] recently proposed *learned branching* that uses supervised machine learning techniques to learn a fast proxy to strong branching. In their work, a function approximating strong branching is first learned in a preliminary phase and is then used within B&B instead of the 'real' strong branching.

In an attempt to push further the ideas proposed by the previous information-based branching strategies, we present an approach that combines the ideas behind learned branching [11] and reliability branching [2]. More specifically, the proposed method uses online learning algorithms to learn a proxy to strong branching (SB). The idea is similar to the one developed by Marcos Alvarez et al. [11] in the sense that the goal is to take SB-like decisions without the computational cost induced by the real SB. However, in this case, the learning is performed in an online manner, during the course of the optimization, which alleviates some important shortcomings of the previous approach. Indeed, since the learning is made online, no preliminary phase is required in order to learn the proxy and no computing time is thus wasted in such a step. Additionally, the learned function is really tailored to the studied problem rendering previous machine learning (ML) concerns obsolete. Moreover, our method uses a reliability mechanism similar to that of reliability branching [2] to decide how the SB score is going to be computed. Technically speaking, our method uses SB scores in order to rank the possible branching variables. The candidate with the largest score is chosen as branching variable. These scores can be computed in two ways: either through the normal SB procedure, or thanks to our learned proxy function. More specifically, if the approximation of the SB score for a candidate variable is deemed unreliable, the real SB score is used, and, conversely, if the approximation is trusted for that variable, the learned proxy is used to generate the score. The proxy function is expected to yield approximate SB scores that are close enough to the real scores, but in a much shorter amount of time than the real SB procedure. The SB proxy is learned in an online fashion because the data used to train the function is generated during the course of B&B. The performed experiments show promising results.

2 Preliminaries

We focus on Mixed-Integer Linear Programming (MILP) problems of the form

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \forall j \in I : x_j \in \{0,1\}, \forall j \in C : x_j \in \mathbb{R}_+, \end{aligned} \tag{1}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$ denote the cost coefficients, the coefficient matrix, and the right-hand side, respectively. I and C are two sets containing the indices of the integer and continuous variables, respectively. We denote the solution at a given node of the B&B by $\boldsymbol{x}^*$, and we will call, with a little abuse, the variable x_i, with $i \in I$, a fractional variable if it has a fractional value in the current solution $\boldsymbol{x}^*$. The set of fractional variables of $\boldsymbol{x}^*$ is denoted F.

MILP problems as defined in Equation (1) are usually solved by the branch-and-bound (B&B) algorithm [10]. B&B builds an optimization tree in which each node represents a version of the initial optimization problem where some integrality constraints of the variables in I are relaxed. Consequently, the problem contained in each node of the tree is called a linear programming relaxation (LP-relaxation) of the initial problem, which is solved with a standard LP optimization algorithm. If the solution found at one node violates some of the initial integrality constraints, i.e., $F \neq \emptyset$, the algorithm creates two nodes, corresponding to two new subproblems. In the case of binary problems, one subproblem is created by adding to the current subproblem one constraint of the form $x_i \leq 0$ and the other subproblem is created by the addition of $x_i \geq 1$, such that variable i is forced to an integral value in the descendants of those nodes. This operation is called branching on variable i. On the other hand, when the solution found at one node respects all initial integrality constraints, i.e., $F = \emptyset$, then the algorithm has found a solution (not necessarily optimal) of the problem and the exploration of that branch of the tree is stopped. Once the tree is entirely explored, the integral solution with the smallest objective value is returned as the optimal solution of the initial problem.

The branching strategy, i.e., the function that chooses a variable i in the set F, is probably the component of B&B that most conditions the efficiency of the algorithm. Indeed, the branching strategy directly influences the number of nodes that the algorithm explores before terminating. This number of nodes needs to be as small as possible so as to minimize the time required to solve the problem, but the time spent choosing the branching variable is also an important aspect that needs to be taken into account. In general, taking a good branching decision, i.e., a decision that minimizes the number of nodes, is time consuming. Consequently, compromises must be made between the time spent choosing a branching variable and the resulting number of nodes of the optimization tree.

In this work, we use supervised machine learning (ML) techniques. In a few words, ML can be used (among other applications) to automatically construct functions from data. Let $D = ((\boldsymbol{\phi}_i, y_i))_{i=1}^N \in (\Phi \times \mathcal{Y})^N$ be a dataset of input-

output pairs, a supervised ML algorithm $\mathcal{A}$ yields a function

$$f^* = \mathcal{A}(D) = \underset{f \in \mathcal{F}}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(y_i, f(\boldsymbol{\phi}_i)),$$

where $\mathcal{F} : \Phi \mapsto \mathcal{Y}$ represents a set of functions that map input values in $\Phi \subset \mathbb{R}^d$ to output values in $\mathcal{Y} \subset \mathbb{R}$. The output f^* is such that a chosen loss function $\mathcal{L}$ is minimized on the input dataset D. In the machine learning community, the inputs $\boldsymbol{\phi}$ of the functions f are called 'features'. Those features are usually not provided as is and they need to be carefully designed for each application.

3 Online learning of strong branching scores

Strong branching (SB) [4] is a very popular branching heuristic that selects a branching variable by explicitly computing the dual bound improvements for each candidate. SB is known to perform very well in terms of the number of nodes of the resulting B&B tree, but requires a very large amount of computational effort to take a decision. In order to alleviate this problem, the method that we propose learns, during the course of the optimization, a function that approximates SB and that can be evaluated more quickly. The approximation of SB is done with a very simple machine learning technique, namely linear regression. Moreover, the method that we propose uses a reliability mechanism that is similar to the one used by reliability branching [2]. Note that applying linear regression to predict a SB score for a given variable requires that the variable be described by a set of features, or explanatory variables. These features need to be carefully designed in order for the method to be effective. We use the same features as in learned branching [11].

Technically speaking, the branching strategy that we propose (olb) is as follows. When B&B needs to branch, a set of candidate variables is selected from the set of fractional variables at the current node and a score is computed for each variable. The variable with the largest score is then selected for branching. These scores can be computed in two ways. On the one hand, if the approximation that we create for the variable is deemed unreliable, the real SB score is computed, together with a set of features describing that variable at the current node. The computed score is used to rank the candidate variable, but it is also stored in a dataset (with the computed features) to improve the approximation of the SB proxy. On the other hand, if the approximation is deemed reliable, the features describing the current candidate are computed and fed to the approximated version of SB, i.e., the function learned with linear regression, in order to quickly generate an approximate SB score, hopefully close to the real SB score.

The real SB scores that are generated when the approximation cannot be trusted are used to train a simple linear regression, whose goal is to predict approximate SB scores. The learning is performed in an online fashion with a simple gradient descent algorithm (using line search) to allow the approximation to evolve during the course of the optimization. In order to determine whether

the approximation for a given variable is reliable or not, we simply count the number of samples (i.e., computed real SB scores and features) that have been generated previously for each variable. This mechanism is comparable to the reliability mechanism of reliability branching [2].

The complete description of our branching strategy is given in Algorithm 1. The proposed method requires four main parameters. First, $\lambda \in \mathbb{N}_0^+$ controls the number of variables that can be considered as branching candidates at each iteration. Second, $\eta \in \mathbb{N}_0^+$ indicates how many samples are required in order to trust the approximation for a specific variable. Lastly, $\delta \in \mathbb{R}_0^+$ and $\sigma \in \mathbb{R}_0^+$ are used to limit the convergence of the gradient descent algorithm in order to avoid undesirable oscillations (cf. Algorithm 1, line 32).

One of the limitations of olb is that there is only a limited number of learning samples per variable. If the dynamics for a variable change, for instance because the tree grows bigger, then the learned linear regression becomes useless, as it does not represent anymore the correct dynamics for that variable. Allowing the linear regression to learn during the entire course of B&B is one way to avoid this issue. We call the resulting method online perpetual learning branching (oplb). The basic mechanisms remain unchanged. The only difference is that, when a branching is actually performed and a real SB score is not generated, some information (the node, the objective value, and the features for the branching variable) is recorded in an ad hoc data structure. When both child nodes created during that branching are explored, the resulting dual bound improvement can be computed exactly. This improvement, which actually corresponds to the real SB score, is then added, together with the stored features, to the learning queue in order to be processed by the learning algorithm. Using this trick allows the branching strategy to adapt over time, even if the dynamics of the problem for a given variable change.

4 Experiments

We assess the efficiency of the proposed approach by comparing different branching strategies on a selection of problems from MIPLIB [6, 3], listed in Table 1. We select only binary problems of small to medium size. The experiments consist in optimizing the selected problems while plugging in different branching strategies. Each problem is optimized 10 times with 10 different random seeds (from 0 to 9). We impose a time limit of 7200 seconds for each optimization.

The experiments are carried out on a 16-core computer, equipped with two Intel Xeon E5520 (2.27GHz, 8 cores) and 32GB RAM, running CPLEX 12.6. We disable presolve in CPLEX but leave the default values for the other parameters (except for the seed). Additionally, we disable parallelism.We compare our approach to three other branching strategies, namely full strong branching (fsb) [4], reliability branching (rb) [2], and learned branching (learned) [11]. The default parameter values $\lambda = 4$ and $\eta = 8$ are used for rb [2]. For fsb, the SB LP-relaxations are solved to optimality, and there is no limit on the number of candidate fractional variables at each node. For learned, we use the default

Algorithm 1 Online learning branching (olb). Note that all variables are assumed to be global.

Inputs: $\boldsymbol{x}^*$ and o are the solution and the objective value at the current B&B node, respectively — $\boldsymbol{w}$ is the weight vector (i.e., the parameters) of linear regression — i is the learning iteration — q is the learning queue — c_L and c_P are data structures that count, respectively, the number of samples already learned and the number of samples waiting in the queue for each variable

Parameters: λ, η, δ, σ

procedure OLB($\boldsymbol{x}^*$, o, $\boldsymbol{w}$, i, q, c_L, c_P)
  best_variable = -1 ; best_score = -1
  for each fractional variable j in $\boldsymbol{x}^*$ **do**
    if number of evaluated candidate variables $> \lambda$ **then**
      break
    end if
    if $c_L(j) + c_P(j) < \eta$ **then** ▷ variable j is unreliable
      $s =$ strongBranchingScore$(j, \boldsymbol{x}^*, o)$ ▷ compute real SB score
      $\boldsymbol{\phi} =$ computeFeatures$(j, \boldsymbol{x}^*, o)$
      $q = q \cup (\boldsymbol{\phi}, s)$ ▷ add score and features to q
      $c_P(j)$ ++
    else ▷ variable j is reliable
      if $c_L(j) < \eta$ **then** ▷ if samples are not learned yet, learn
        LEARN($\boldsymbol{w}$, i, q, c_L, c_P)
      end if
      $\boldsymbol{\phi} =$ computeFeatures$(j, \boldsymbol{x}^*, o)$ ▷ compute features $\boldsymbol{\phi}$ for variable j
      $s = \boldsymbol{w}^\top \boldsymbol{\phi}$ ▷ compute approx. SB score for j
    end if
    if $s >$ best_score **then**
      best_variable = j ; best_score = s
    end if
  end for
  if size(q) $>$ maximum allowed size **then** ▷ limit the size of the queue
    LEARN($\boldsymbol{w}$, i, q, c_L, c_P)
  end if
  return best_variable
end procedure
procedure LEARN($\boldsymbol{w}$, i, q, c_L, c_P)
  for $(\boldsymbol{\phi}, s) \in q$ **do**
    $l = s - \boldsymbol{w}^\top \boldsymbol{\phi}$ ▷ current loss
    $\nabla l = -l\boldsymbol{\phi}$ ▷ current gradient of the loss
    $\boldsymbol{w} = \boldsymbol{w} - \exp(-\delta \lfloor i\sigma^{-1} \rfloor) \frac{-l}{\boldsymbol{\phi}^\top \nabla l} \nabla l$ ▷ update equation
    $i = i + 1$
  end for
  $q = \emptyset$ ▷ clear learning queue
  $\forall j : c_L(j) = c_L(j) + c_P(j); c_P(j) = 0$ ▷ increase reliability of all variables
end procedure

Table 1. List of problems (44) from MIPLIB3 [6] and MIPLIB2003 [3].

10teams	aflow30a	aflow40b	air03	air04	air05	cap6000	dcmulti
egout	fiber	fixnet6	harp2	khb05250	l152lav	lseu	mas74
mas76	misc03	misc06	misc07	mitre	mod008	mod010	mod011
modglob	nw04	opt1217	p0033	p0201	p0282	p0548	p2756
pk1	pp08a	pp08aCUTS	qiu	rentacar	rgn	set1ch	stein27
stein45	tr12-30	vpm1	vpm2				

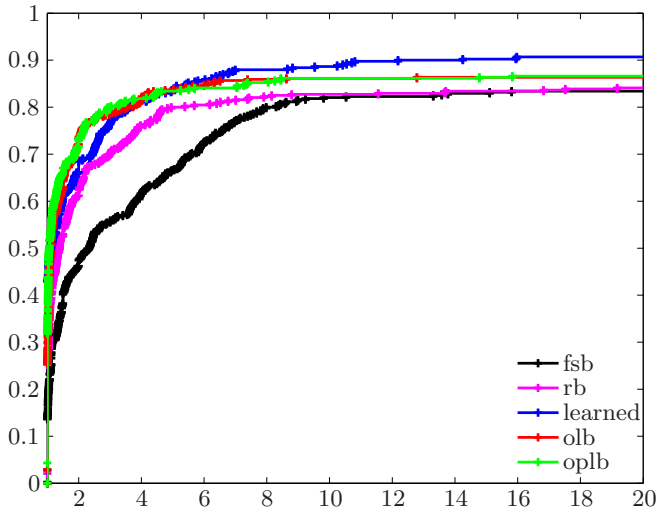
parameters [11], i.e., $N = 100$, $k = |\phi|$, and $n_{\min} = 20$. The parameters for our methods, online learning branching (olb) and online perpetual learning branching (oplb), are $\lambda = 4$, $\eta = 8$, $\delta = 0.01$, and $\sigma = 500$.

We use performance profiles [7] to compare the different approaches. The performance profiles are drawn considering that the pairs composed of a problem and a seed are a single problem. Consequently, the performance profiles are constructed with 440 problems (every combination of problem and seed). Figure 1(a) illustrates the performance of the methods in terms of time, while Figure 1(b) focuses on the number of nodes required to solve the problems. The results show that fsb behaves as expected: very good in terms of nodes, but rather slow in general. The proposed method (olb, oplb) compares favorably to the other methods. It takes rather good branching decisions and is shown to be very fast in the beginning of the optimization (better than the others when the time ratio is less than $\approx$3.5). Learned branching is dominated by olb and oplb in the beginning but catches up quite quickly[1] (the ratio of solved instances exceeds the other methods when a large enough amount of time is provided). Learned is also very effective in terms of nodes. Overall, rb is dominated by the other approaches both in terms of time and in terms of nodes. Finally, we note that learning perpetually (oplb) does not improve significantly the performance of the normal online learning approach (olb).

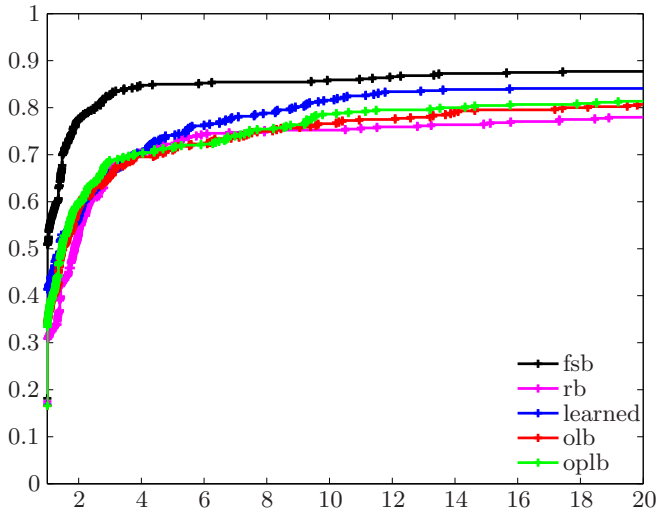
5 Conclusion

We propose a branching strategy based on online machine learning algorithms that is similar in its mechanisms to reliability branching. The method is meant to work with a supposedly good branching strategy. During the branching procedure, if a candidate variable is deemed unreliable, the good branching strategy is used, which increases the confidence level of the candidate variable. At some point in time, the variable can be trusted and, from that moment on, an approximation of the score generated by the good strategy is used instead of the real score. The real score is approximated using online supervised learning. We focus on strong branching, but the proposed approach can be applied to any branching strategy. Our results show that our approach compares favorably to other branching strategies both in terms of time and in terms of nodes.

[1] It is to be noted that the time taken to train the 'learned' branching strategy is not taken into account in the performance profiles.



(a) Performance profile in terms of time



(b) Performance profile in terms of nodes

Fig. 1. Performance profiles for the competing methods. The performance profiles report the probability that a solver solves a problem vs. a performance ratio. A point (x, y) on a performance profile curve should be understood as 'there is a probability y that the method solves a problem if it is given at most x times as much budget as the best solver needs to solve the problem'. The leftmost part of the performance profiles indicate how good the solvers are, i.e., how fast they solve the problems in terms of the chosen metric (time or number of nodes, in this case). On the other hand, the rightmost part of the graphs are usually an image of the robustness of a method, i.e., this part indicates the proportion of problems that will eventually be solved by a method.

Bibliography

[1] T. Achterberg and R. Wunderling. Mixed integer programming: analyzing 12 years of progress. In *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

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

[3] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):361–372, 2006.

[4] D. Applegate, R.E. Bixby, V. Chvátal, and W. Cook. Finding cuts in the tsp (a preliminary report). Technical Report 05, DIMACS, 1995.

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

[6] R.E. Bixby, S. Ceria, C. McZeal, and M.W.P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0, 1996.

[7] E.D. Dolan and J.J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[8] M. Fischetti and M. Monaci. Backdoor branching. *INFORMS Journal on Computing*, 25(4):693–700, 2012.

[9] F.K. Karzan, G.L. Nemhauser, and M.W.P. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1(4):249–293, 2009.

[10] A.H. Land and A.G. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[11] A. Marcos Alvarez, Q. Louveaux, and L. Wehenkel. A supervised machine learning approach to variable branching in branch-and-bound. 2014.