Submitted to *INFORMS Journal on Computing*
manuscript (Please, provide the mansucript number!)

A Machine Learning-Based Approximation of Strong Branching

Alejandro Marcos Alvarez, Quentin Louveaux, Louis Wehenkel
Université de Liège, Department of EE&CS,
Sart-Tilman B28, Liège, Belgium, {amarcos,q.louveaux,l.wehenkel}@ulg.ac.be

We present in this paper a new generic approach to variable branching in branch-and-bound for mixed-integer linear problems. Our approach consists in imitating the decisions taken by a good branching strategy, namely strong branching, with a fast approximation. This approximated function is created by a machine learning technique from a set of observed branching decisions taken by strong branching. The philosophy of the approach is similar to reliability branching. However, our approach can catch more complex aspects of observed previous branchings in order to take a branching decision. The experiments performed on randomly generated and MIPLIB problems show promising results.

Key words: branch-and-bound; variable branching; strong branching; supervised machine learning

1. Introduction

Most Mixed-Integer Programming (MIP) solvers are based on the branch-and-bound (B&B) algorithm (Land and Doig 1960). Over the years, numerous fundamental features, such as cutting planes, presolve, heuristics or advanced branching strategies, have been added to the solvers in order to improve their performances (Achterberg and Wunderling 2013). However, among those additional features, branching, i.e., the process that divides the feasible region into two or more subproblems, is probably the key component that most affects the efficiency of the solver (Achterberg and Wunderling 2013).

Branching strategies have been extensively studied in the literature, and we briefly review here some key contributions to that field. The simplest criterion, known as *most-infeasible branching*, consists in branching on the variable that has the greatest fractional part, i.e., the variable whose fractional part is closest to 0.5. However, most-infeasible branching is known to perform poorly in practice, and other methods, such as *pseudocost branching*

(Benichou et al. 1971), have been developed later. Pseudocost branching keeps a history of the dual bound increases observed during previous branchings, and uses this information to estimate the dual bound improvements for each candidate variable at the current node. Although pseudocost branching is very efficient in terms of computation time, the branchings performed at the very beginning of the B&B tree might be inefficient, as no reliable history has been recorded at that time. Later, Applegate et al. (1995) proposed a strategy, known as *strong branching*, that overcomes this limitation. Strong branching explicitly evaluates the dual bound increase for each fractional variable by actually computing the LP relaxations resulting from the branching on that variable. The variable that leads to the largest increases is chosen as branching variable for the current node. Despite its apparent simplicity, strong branching is, up to now, the most efficient branching strategy in terms of the number of nodes in the B&B tree. However, this efficiency is achieved at the expense of computation time, and strong branching is unfortunately intractable in practice. More recently, Achterberg et al. (2005) proposed to combine the advantages of both pseudocost and strong branching in a branching strategy called *reliability branching*. Many other branching strategies have been developed for the past 15 years, such as *inference branching* (Li and Anbulagan 1997), *non-chimerical branching* (Fischetti and Monaci 2012b), *active constraint branching* (Patel and Chinneck 2007), and *cloud branching* (Berthold and Salvagnin 2013), but their thorough description is beyond the scope of this paper. Finally, let us mention *hybrid branching* (Achterberg and Berthold 2009), which is probably today's state-of-the-art branching strategy. Hybrid branching efficiently combines five scores obtained from other common branching strategies (including reliability branching as well as other strategies used in CSP and SAT solving), and is used as the main branching strategy in CPLEX 12.5 (Achterberg and Wunderling 2013). More specifically, hybrid branching uses the considered branching strategies to compute five different scores for each candidate variable. These scores are first normalized and then merged into a single value through a weighted sum. The variable that maximizes that sum is chosen as branching variable.

Following the ideas introduced by pseudocost branching, researchers have recently started investigating branching strategies that rely on information collected through multiple B&B restarts. *Backdoor branching* (Fischetti and Monaci 2012a) and *information-based branching* (Karzan et al. 2009) are two key contributions to this aspect. The mechanism of these strategies is two-phased. During the first phase, the optimization of the current problem is restarted from the beginning multiple times (using several heuristics to guide the search), and the algorithm collects some information about each run (e.g., the conflict clauses or the most important branching variables). In the second phase, the real optimization starts, and the harvested information is used to take efficient branching decisions. The idea behind those methods is to quickly and superficially explore different parts of the B&B tree and to decide, based on those shallow explorations, which part it is best to focus on. The work presented in this paper relies on the very same basic idea. Finally, let us mention the work of Di Liberto et al. (2013) who recently proposed a method that uses machine learning techniques to switch between several branching heuristics. More specifically, they present an approach that first determines which strategy, among a restricted set of branching heuristics, produces the best results for a given set of problem instances. Then, when a new problem is solved by B&B, the algorithm checks, at several points in time, which strategy should be used at that particular moment, based on the optimal strategies that were found in the first phase. Although the main mechanism of their approach is different, the ideas are very similar, and their method has strong ties with the one developed in this paper. Indeed, their method also relies on the idea of leveraging prior information through principled learning techniques, but, while they use machine learning to switch between existing branching strategies, we use machine learning to provide a quick approximation to an existing branching strategy.

Our goal is to overcome the large computational overhead resulting from a strong branching decision. Speeding up strong branching-like decisions is not a new idea, as it is already behind other branching heuristics, such as reliability branching or non-chimerical branching. In this paper, we propose an alternative approach that uses machine learning techniques in order to imitate the strong branching decisions in an efficient way. More specifically, we propose a two-phased approach that yields a ‘learned’ branching strategy that can be used within B&B as an approximation of strong branching. The first phase involves solving a set of training problems with strong branching as branching heuristic in order to generate a set of branching decisions. During this phase, each branching decision is recorded in a dataset, called training set, that will then be used by a machine learning algorithm to learn a function imitating strong branching decisions. In the second phase, we introduce, as any other branching heuristic, the learned heuristic into B&B, and evaluate its efficiency on a set of test problems. An important characteristic of our approach is that the first phase needs only to be done once. Indeed, once the branching heuristic is learned from the training set, it can be included directly into B&B, without requiring a new training phase each time a problem needs to be solved. In that sense, we can say that the training phase can be done in an off-line fashion, thus avoiding useless computational overhead at the beginning of each optimization.

In this paper, we address binary Mixed-Integer Linear Programming (MILP) problems of the form

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

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ respectively denote the cost coefficients, the coefficient matrix and the right-hand side. 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 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 x^*. The set of fractional variables of x^* is denoted F.

2. Preliminaries

In this section, we first present the concept of branching in a general functional form, and then briefly introduce the field of machine learning for the beginner.

2.1. Functional form of branching strategies

Any branching heuristic can be formulated in a generic functional form $\mathcal{B}$ such that

$$\mathcal{B} : (i, \cdot) \mapsto \mathbb{R},$$

where i represents the index of the candidate branching variable, and $\cdot$ represents unspecified arguments of $\mathcal{B}$. The branching variable i^* is chosen as the one that maximizes[1] the scores given by $\mathcal{B}$, i.e.,

$$i^* = \arg\max_{i \in F} \mathcal{B}(i, \cdot).$$

[1] If ties occur, the variable that arrives first in the lexicographical order, i.e., the one with the smallest i, is chosen.

The functional form $\mathcal{B}$ is different for every branching criterion, and proposing a new branching heuristic merely consists in providing a new $\mathcal{B}$, including its implementation and the specification of its arguments. For example, in the case of most-infeasible branching (MIB), $\mathcal{B}_{\text{mib}}$ only requires the current fractional solution to output a score for a variable. Then, the functional form of MIB is written $\mathcal{B}_{\text{mib}}(i, x^*) = \min(1 - x_i^*, x_i^*)$. Another example is strong branching, which requires more input arguments, as it needs more information to take a decision. The functional form of strong branching (SB) is $\mathcal{B}_{\text{sb}}(i, c, A, b, x^*, l^*, u^*)$, where l^* and u^* represent the lower and upper bounds of the variables at the current node, respectively. The implementation of $\mathcal{B}_{\text{sb}}$ consists in creating two subproblems by changing the upper and lower bounds of variable i in the current problem to $\lfloor x_i^* \rfloor$ and $\lceil x_i^* \rceil$, respectively. The LP-relaxations of the subproblems thus created are then solved, and, for each subproblem, the difference between the objective value of the subproblem and the current problem is computed. These differences represent the objective increases observed between the current node and the subproblems when tighter bounds are used for the variable i. The output of $\mathcal{B}_{\text{sb}}$ is finally given by the product of the computed differences.

2.2. Learning functions from data

A very popular use of supervised machine learning (ML) concerns the automatic construction, or learning, of functions from data. Basically, ML is used in the context of some task $\mathcal{T}$, for which a function f mapping the current state of $\mathcal{T}$ to a given output space is needed. Such a function maps inputs from a space Φ to an output space $\mathcal{Y}$, i.e., $f(\cdot) \in \mathcal{F} : \Phi \mapsto \mathcal{Y}$, where $\mathcal{F}$ is the set of possible mappings from Φ to $\mathcal{Y}$. Formally, a general machine learning algorithm $\mathcal{A}$ is a procedure of the form $\mathcal{A} : (\Phi \times \mathcal{Y})^N \mapsto \mathcal{F}$ that takes as input a dataset $D = ((\phi_i, y_i))_{i=1}^N \in (\Phi \times \mathcal{Y})^N$, and that outputs a function $f \in \mathcal{F}$ that minimizes some loss function $\mathcal{L}$ on the dataset D, where N represents the number of elements in the dataset. Stated in mathematical terms, the function f^* resulting from the application of algorithm $\mathcal{A}$ to dataset D is given by

$$f^* = \mathcal{A}(D) \in \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} \mathcal{L}(y_i, f(\phi_i)) .$$

Ideally, the input of function f should be the state of $\mathcal{T}$ itself. However, representing the complete state is often a difficult problem, e.g., because its dimensionality is too high, or because it contains a lot of irrelevant information. For this reason, in the machine learning community, the inputs ϕ of the functions f are usually 'features' representing a simplified version of the state of $\mathcal{T}$. Formally, features are characteristics extracted from the state of $\mathcal{T}$, such that those features represent part of the current state, ideally the part that most influences the output. The features, which are typically designed by human experts, often determine the efficiency of learning methods. As they represent only part of the current state of the task, it is important that the parts described by the features are indeed correlated with the desired output. Note that in this context, 'correlated' should not be understood in the traditional sense of the Pearson or Spearman correlation between individual features and the desired output. It rather means that it is crucial, for the learning algorithm to work, that there exists a strong enough statistical dependency, possibly taking very complex forms, between the set of chosen features and the desired output. For this reason, the features need to be carefully designed and tailored to the problem of interest.

3. Learning branching decisions

Since machine learning techniques can be used to approximate (learn) functions, and since branching can be determined by a function, it seems natural to consider machine learning as a reasonable way to build a branching strategy. We must stress here that machine learning does not provide a completely new branching criterion, as it requires the observation of real branching decisions to actually build a branching function. Rather than providing a new branching heuristic, machine learning constructs a branching strategy that imitates the decisions of the branching strategy that generated the dataset D, in our case, strong branching.

Our goal is to create an efficient approximation of strong branching that could be used in practice. In other words, the branching heuristic $\mathcal{B}_{\text{learned}}(i, \phi_i)$ that we propose is such that

$$\mathcal{B}_{\text{learned}}(i, \phi_i) \approx \mathcal{B}_{\text{sb}}(i, c, A, b, x^*, l^*, u^*), \tag{2}$$

where ϕ_i is a feature vector describing the state of the optimization problem at the current node from the perspective of variable i. The feature vector ϕ_i does not describe the current node of the B&B tree, but rather describes variable i in the current node. Those features need to be efficiently computable and have to well represent the problem at the current B&B node from the perspective of variable i. Section 4 explains in more detail how the features are designed.

3.1. Dataset generation

The first step towards the creation of $\mathcal{B}_{\text{learned}}$ with a machine learning technique is to generate a dataset from which the function can be learned.

In order to create a dataset of pairs (ϕ_i, y_i), we optimize the problems contained in some sets, which we call training problems, with B&B, using strong branching as branching heuristic. At each node explored by B&B during the optimization of those problems, the strong branching score $\mathcal{B}_{\text{sb}}(i, c, A, b, x^*, l^*, u^*) = s_i$ is computed for each fractional variable $i \in F$ together with the features ϕ_i associated with that variable. These features-score pairs are then saved in the dataset D_{sb}, which is then used as input of the learning algorithm to generate a learned branching strategy $\mathcal{B}_{\text{learned}}(i, \phi_i)$.

3.2. Machine learning algorithm

Once the features are designed to correctly describe each variable of the problem, and once a dataset of input-output pairs is available, we can apply a machine learning algorithm to learn a function from the dataset. In this work, we use *Extremely Randomized Trees* (Geurts et al. 2006), or ExtraTrees. ExtraTrees are based on an ensemble of regression trees and are a slightly modified version of the better known random forests (Breiman 2001). Our choice is motivated by the simplicity and the robustness of ExtraTrees. Indeed, the performance of ExtraTrees is very robust against the choice of their parameters, and the default values provided in (Geurts et al. 2006) work very well in practice.

4. Features describing the current subproblem

As mentioned earlier, the features are the key component of our approach, since they critically condition the efficiency of the method. On the one hand, the features need to be complete and precise in order to describe the subproblem as accurately as possible. On the other hand, they need to be efficient to compute. It is important to keep this tradeoff in mind, because there are many good features that could have a very positive impact on the efficiency of the method, but that are too expensive to compute. An example of such features is the objective increase obtained when branching is performed on a variable, i.e., the numbers that are actually used by strong branching to take a decision. Such features cannot be used in our approach, because of the huge computational overhead required by their computation.

Before describing the features we used, we need to emphasize the three properties that these features should have. First, the number of features needs to be independent of the size of the problem instance. Indeed, the learning algorithm that we selected can cope only with datasets in which all the feature vectors ϕ_i have the same number of elements. If this number depends on the size of the problem, a different branching strategy must be learned for each problem size. This is of course an impractical situation, and enforcing the size-independency is the best way to obtain a single learned branching strategy that can be used for any problem size. This might seem a straightforward requirement, but the size-independency is not trivial to achieve. Elementary features such as c, A or b cannot be used directly in that case. Another important requirement is that the features should be invariant with respect to irrelevant changes in the problem, such as row or column permutation. Finally, the developed features need to be independent of the scale of the problem, i.e., if the parameters (c, A, and b) are multiplied by some factor, the features should remain identical.

The features ϕ_i that we describe assume that the problem is in the canonical form (1). Each feature vector ϕ_i is computed for variable i at the current node, before being fed to $\mathcal{B}_{\text{learned}}$. The features are divided into three subsets representing different aspects of the optimization state, namely ‘static problem features’, ‘dynamic problem features’, and ‘dynamic optimization features’.

Note that, Hutter et al. (2014) recently introduced a certain number of features describing MIP problems that are related to the ones presented in this work. Some of our features are based on similar ideas as theirs. Specifically, features related to A_{ji}/b_j, $A_{ji}/\sum A_{ji}$, and the slack variables are used in both studies. There is one major difference however in the design and use of those features. In Hutter et al. (2014)'s work, the features are aggregated (either over the constraints or over the variables), because they are used to characterize a problem instance as a whole. In our work, on the other hand, the features are not aggregated over the variables, because they are meant to characterize a single variable within the problem. The fact that the design choices are different is not surprising. Indeed, since the use of ML in this paper and in Hutter et al. (2014)'s paper is in pursuit of different goals, the features that are relevant for each task are likely to be different.

4.1. Static problem features

The first set of features are computed from the sole parameters c, A and b. They are calculated once and for all and they represent the static state of the problem. Their goal is to give an overall description of the problem. These features are designed such that the aforementioned requirements are fulfilled.

The first three of them are devoted to the description of the current variable in terms of the cost function. Besides the sign of the element c_i, we also use $|c_i| / \sum_{j:c_j \geq 0} |c_j|$ and $|c_i| / \sum_{j:c_j<0} |c_j|$. Distinguishing both is important, because the sign of the coefficient in the cost function is of utmost importance for evaluating the impact of a variable on the objective value.

The second class of static features is meant to represent the influence of the coefficients of variable i in the coefficient matrix A. We develop three measures, namely M_j^1, M_j^2 and M_j^3, that describe variable i within the problem in terms of the constraint j. Once the values of the measure M_j are computed, the corresponding features added to the feature vector ϕ_i are given by $\min_j M_j$ and $\max_j M_j$. The rationale behind this choice is that, when it comes to describing the constraints of a given problem, only the extreme values are relevant.

The first measure M_j^1 is composed of two parts: M_j^{1+} computed by $A_{ji}/|b_j|$, $\forall j$ such that $b_j \geq 0$, and M_j^{1-} computed by $A_{ji}/|b_j|$, $\forall j$ such that $b_j < 0$. The minimum and maximum values of M_j^{1+} and M_j^{1-} are used as features, to indicate by how much a variable contributes to the constraint violations.

Measure M_j^2 models the relationship between the cost of a variable and the coefficients of the same variable in the constraints. Similarly to the first measure, M_j^2 is split in $M_j^{2+} = |c_i| / A_{ji}$, $\forall j$ with $c_i \geq 0$, and $M_j^{2-} = |c_i| / A_{ji}$, $\forall j$ with $c_i < 0$. As for the previous measure, the feature vector ϕ_i contains both the minimum and the maximum values of M_j^{2+} and M_j^{2-}.

Finally, the third measure M_j^3 represents the inter-variable relationships within the constraints. The measure is split into $M_j^{3+} = |A_{ji}| / \sum_{k:A_{jk} \geq 0} |A_{jk}|$ and $M_j^{3-} = |A_{ji}| / \sum_{k:A_{jk}<0} |A_{jk}|$. M_j^{3+} is in turn divided in M_j^{3++} and M_j^{3+-} that are calculated using the formula of M_j^{3+} for $A_{ji} \geq 0$ and $A_{ji} < 0$, respectively. The same splitting is performed for M_j^{3-}. Again, the minimum and maximum of the four M_j^3 computed for all constraints are added to the features.

4.2. Dynamic problem features

The second type of features is related to the solution of the problem at the current B&B node. Those features contain the proportion of fixed variables at the current solution, the up and down fractionalities of variable i, the up and down Driebeek penalties (Driebeek 1966) corresponding to variable i, normalized by the objective value at the current node, and the sensitivity range of the objective function coefficient of variable i, also normalized by $|c_i|$.

4.3. Dynamic optimization features

The last set of features is meant to represent the overall state of the optimization. These features summarize global information that is not available from the single current node. When branching is performed on a variable, the objective increases are stored for that variable. From these numbers, we extract statistics for each variable: the minimum, the maximum, the mean, the standard deviation, and the quartiles of the objective increases. These statistics are used as features to describe the variable for which they were computed. As those features should be independent of the scale of the problem, we divide each objective increase by the objective value at the current node, such that the computed statistics correspond to the relative objective increase for each variable. Finally, the last feature added to this subset is the number of times variable i has been chosen as branching variable, normalized by the total number of branchings performed.

5. Experiments

This section describes the experimental procedure that we set up in order to assess the efficiency of our approach. It is composed of three steps: (1) we generate a dataset D_{sb} using strong branching, (2) we learn from it a branching heuristic, and (3) we compare the learned heuristic with other branching strategies on various problems. This section describes the different datasets used within our approach as well as the experimental setup.

5.1. Problem sets

We use two types of problem sets, namely randomly generated problem sets and standard benchmark problems from the MIPLIB (Bixby et al. 1996, Achterberg et al. 2006). The random problem sets are used for both training (steps 1 and 2) and assessing (step 3) the heuristics, whereas MIPLIB problems are only used for assessment (step 3). The reasons for using random problems are two-fold.

First, the typical evaluation procedure in machine learning consists in evaluating a function learned from a dataset on a different dataset. If the function is both learned and evaluated on the same dataset, the estimated performance might be too optimistic and might thus not reflect the overall performance of the learned function. To prevent this, the datasets are separated into two parts: the first part is used for training and the second part is used for assessing the method. Since the MIPLIB is the traditional evaluation benchmark for MIP methods, and in order to comply with the machine learning assessment methodology, we decided to use other problems, i.e., the randomly created problems, to learn our branching heuristic. Note that it would have been possible to use other techniques, such as k-fold cross validation, to correctly evaluate the performance of different branching strategies on the MIPLIB without requiring new problems to be created.

Secondly, the performance of any machine learning procedure increases with the size and the variety of the dataset used by the learning algorithm. In theory, the expected accuracy of the ML procedure over the entire input space of the learned function is a monotonically increasing function of the size of the dataset D. As this dataset is created by optimizing a set of problems, increasing the number of problems in the problem set yields a bigger dataset D. It is thus to our advantage to consider as many problems as possible to create D.

The problems used in this work are rather small. The small problem size used in our experiments is justified by the need to observe the entire B&B tree in the dataset D. Indeed, the learned branching heuristic can only reflect the branching decisions found in the dataset. If the training problems are too big or too difficult to solve, the dataset will only contain branching decisions observed at the beginning of the B&B tree, and will not reflect all the possible branching decisions. It is thus important that the dataset also contains branching decisions from the very bottom of the B&B tree. This consideration has to be taken into account when choosing the problems to include in our problem sets. For this reason, the size of the training problems has to be limited, so that the problems can be solved as much as possible with strong branching in a reasonable amount of time. Similarly, since the size of the training problems is limited, so is the size of the test problems.

5.1.1. Random problems. We randomly generate three sets of binary-integer or mixed binary-integer minimization problems that each contain two different types of constraints. The possible constraints are chosen among set covering (SC), multi-knapsack (MKN), bin packing (BP), and equality constraints (EQ). We generated problems that contain constraints of type BP-EQ, BP-SC, and MKN-SC. The number of variables, the number of constraints, and the values of the elements in the matrices c, A and b are randomly generated. More specifically, we arbitrarily choose some bounds on the number of variables,

Table 1 Randomly generated problem sets. 'all', 'bin' and 'cont' indicate the total number, and the number of binary and continuous variables in the problems, respectively.

		Variables					
		all		bin		cont	
	# prob.	min	max	min	max	min	max
BPEQ_train	25	201	225	150	179	43	54
BPEQ_test	50	193	234	145	185	43	54
BPSC_train	25	109	136	109	136	0	0
BPSC_test	50	109	137	109	137	0	0
MKNSC_train	25	188	358	188	358	0	0
MKNSC_test	50	185	342	185	342	0	0

Table 2 Randomly generated problem sets. 'all', 'EQ', 'BP', 'SC' and 'MKN' specify the total number, and the number of equality, bin packing, set covering, and multi-knapsack constraints in the problem sets, respectively.

	Constraints									
	all		EQ		BP		SC		MKN	
	min	max	min	max	min	max	min	max	min	max
BPEQ_train	94	138	39	50	55	89	0	0	0	0
BPEQ_test	94	135	39	50	53	89	0	0	0	0
BPSC_train	80	110	0	0	50	73	28	40	0	0
BPSC_test	80	112	0	0	50	75	27	39	0	0
MKNSC_train	108	156	0	0	0	0	61	77	42	84
MKNSC_test	108	160	0	0	0	0	58	77	43	89

Table 3 List of problems from MIPLIB3 and MIPLIB2003.

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				

on the number of constraints, and on the elements contained in the matrices c, A and b. Then, we generate the parameters defining the problem with a distribution that is uniform between the chosen bounds.

The number of variables in these problems is of the order of a couple of hundreds, and the number of constraints is of the order of one hundred. As some of those problems are going to be used to generate the training dataset, we randomly split each family into a 'train' and a 'test' set. In the end, we have six datasets 'BPEQ_train', 'BPEQ_test', 'BPSC_train', 'BPSC_test', 'MKNSC_train' and 'MKNSC_test'. The test sets contain 50 problems each, while the train sets each contain 25. Tables 1 and 2 summarize statistics about the randomly generated problems. More specifically, those tables respectively contain the bounds on the number of variables and on the number of constraints of the problems belonging to each randomly generated set. Remember that, as explained in the previous section, the generated problems are intentionally kept small. Those datasets are available online[2], or can be provided upon request.

5.1.2. MIPLIB. We also compare different branching strategies on a set composed of problems from MIPLIB3 (Bixby et al. 1996) and MIPLIB2003 (Achterberg et al. 2006). We eliminated the non-binary problems and the problems that were too big in order to match the size of the randomly generated problems. The set finally contains 44 problems. Table 3 lists the test problems taken from MIPLIB3.0 and MIPLIB2003.

5.2. Experimental setup

The computations have been performed on a 16-core computer, equipped with two Intel Xeon E5520 (2.27GHz, 8 cores, and 8MB cache) and 32GB RAM, running CentOS 5.4 and CPLEX 12.2. We ran our experiments on our selection of the MIPLIB problems and on the problem sets 'BPEQ_test', 'BPSC_test' and 'MKNSC_test', described in Section 5.1. The training dataset D_{sb} is generated from the problem sets 'BPEQ_train', 'BPSC_train' and 'MKNSC_train'.

To assess only the performance of the different branching strategies, we disable heuristics, cuts, and presolve in CPLEX (except for the last experiment). For each optimization, only one core is made available, so that parallelism is disabled as well. We compare our approach to five other branching strategies, namely random branching (random), most-infeasible branching (MIB), non-chimerical branching (NCB) (Fischetti and Monaci 2012b), full strong branching (FSB), and reliability branching (RB) (Achterberg et al. 2005). Random branching is a branching strategy in which the variable is randomly chosen among the fractional variables. We use the perseverant version of non-chimerical branching (Fischetti and Monaci 2012b), and the default parameter values $\lambda = 4$ and $\eta = 8$ for reliability branching (Achterberg et al. 2005). Moreover, the strong branching LP-relaxations are solved to optimality, and there is no limit on the number of candidate fractional variables at each node.

The dataset D_{sb} contains around 7×10^7 training examples. This number is too large and we thus use only 10^5 of them, randomly selected without replacement from the original dataset D_{sb}. The chosen parameters of ExtraTrees are $N = 100$, $k = |\phi|$, and $n_{\min} = 20$. More details about the parameters of the ExtraTrees can be found in the supplemental

[2] `http://www.montefiore.ulg.ac.be/~ama/research.php`

material. In this setup and on the considered computers (without parallelization), training the branching heuristic takes around 6 minutes. Naturally, this time can be amortized if the same heuristic is used to optimize several problems and even reduced by slightly changing the parameters of the ExtraTrees (at the potential expense of performance). In this work, we use only a fraction of the dataset that we generate in order to train a model that approximates strong branching. This choice is motivated by the fact that the computational efficiency of ExtraTrees is heavily dependent on the size of the training set. The more samples there are in the set, the longer it takes to predict a value. This conflicts with the speed requirements of our method. There exist other learning techniques that do not suffer from this limitation and for which more data could be used without impacting the prediction speed. Note that including more data is beneficial to the prediction accuracy, but, in our case, the gain in the accuracy of the strong branching estimates is not sufficient to counterbalance the additional time taken to make a prediction.

Two types of experiments are performed: one where the optimization is stopped early based on a limit either on the number of explored nodes or on the time spent, and another one where the problems are solved until optimality however long it takes. For the first experiment, the rationale behind the two considered limits is to evaluate different aspects of the branching strategies. When the optimization is limited by the number of explored nodes, we can compare the branching strategies based on the closed gap[3] and on the time spent to actually explore that given number of nodes. This sheds some light on how good a branching strategy is compared to other branching strategies. In these conditions, FSB is usually the best in terms of closed gap and the worst in terms of time spent. On the other hand, the time limit is useful to assess different strategies in practical conditions where the number of nodes matters less than the time required to solve a problem. The closed gap is also used in that experiment to assess how far from the optimum the optimization is after a given amount of time. In this case, FSB is typically outperformed, in terms of closed gap, by other strategies. The second experiment (without limit) is used to assess all branching strategies in a practical situation where the problems need to be solved to optimality.

[3] The closed gap ($\in [0;1]$) is the ratio of the difference between the current dual bound and the objective value of the initial LP-relaxation, to the difference between the optimal objective value and the objective value of the initial LP-relaxation. A value close to 1 indicates that the optimization is almost finished.

5.3. Results

We now present a selection of results comparing our approach to other branching strategies (random, MIB, NCB, FSB, and RB). Tables 4, 5, and 7 report these results. In these tables, 'Cl. Gap' refers to the closed gap, and 'S/T' indicates the number of problems solved within the provided nodes or time limit, versus the total number of problems. 'Nodes' and 'Time' respectively represent the number of explored nodes and the time spent (in seconds) before the optimization either finds the optimal solution, or stops earlier because of one stopping criterion. Those values are measured separately for each problem and are averaged in the tables.

5.3.1. Experiments with limits. Table 4 first shows the results achieved on the random test problem sets for both stopping criteria. Those results show that our approach succeeds in efficiently imitating FSB. Indeed, the experiments performed with a limit on the number of nodes show that the closed gap is only 9% smaller, while the time spent is reduced by 85% compared to FSB. The experiments with a time limit show that the reduced time required to take a decision allows the learned strategy to explore more nodes, and to thus further close the gap than FSB. While these results are encouraging, they are still slightly worse than the results obtained with RB, which is both closer to FSB and faster than our approach.

In addition to the previously presented branching strategies, Table 4 contains one extra experiment for each family of random problems. The normal learned branching strategy, i.e., ($n_{\min} = 20$, all), is learned based on a dataset containing samples from the three types of training problems, i.e., BPSC, BPEQ, and MKNSC. We also investigate, for each type of random problem, the effect of training the branching strategy from a dataset generated with training problems of the same type as the target test problems. In other words, when we test this strategy on the BPSC test problems, the branching rule is learned based on samples generated with BPSC train problems only, which is indicated in the table by ($n_{\min} = 20$, BPSC only). Overall, the results show that the strategies learned only on the type of problem on which they are tested perform a bit better than the strategy learned from a dataset containing a mixture of the three types of problems. This indicates that the approach can benefit from training on a specific type of problem, and that the performance of the learned branching policy improves when the problems to optimize are aligned with the problems that are used to generate the training dataset.

Table 4 Optimization results for the random problems.

	Node limit (10^5 nodes)			Time limit (10 min.)		
BPSC_test problems	S/T	Cl. Gap	Time (s)	S/T	Cl. Gap	Nodes
Random	0/50	0.36	6.73	0/50	0.65	879,696
MIB	0/50	0.39	7.02	0/50	0.68	836,675
NCB	0/50	0.54	117.45	0/50	0.66	45,048
FSB	0/50	0.55	243.70	0/50	0.63	27,157
RB	0/50	0.52	28.29	1/50	0.77	223,369
Learned ($n_{\min} = 20$, all)	0/50	0.48	56.36	0/50	0.67	112,918
Learned ($n_{\min} = 20$, BPSC only)	0/50	0.51	60.54	0/50	0.70	109,066
BPEQ_test problems						
Random	0/50	0.33	17.44	0/50	0.55	366,982
MIB	0/50	0.40	17.27	0/50	0.61	368,309
NCB	0/50	0.81	290.49	0/50	0.86	22,605
FSB	0/50	0.83	681.75	0/50	0.82	9,492
RB	0/50	0.80	74.53	10/50	0.95	90,273
Learned ($n_{\min} = 20$, all)	0/50	0.75	77.97	5/50	0.92	106,057
Learned ($n_{\min} = 20$, BPEQ only)	0/50	0.77	85.68	4/50	0.92	86,370
MKNSC_test problems						
Random	0/50	0.56	7.26	24/50	0.95	587,123
MIB	0/50	0.60	7.39	31/50	0.97	496,475
NCB	0/50	0.67	102.23	5/50	0.83	51,749
FSB	0/50	0.68	135.41	5/50	0.83	46,832
RB	0/50	0.65	27.79	18/50	0.94	173,513
Learned ($n_{\min} = 20$, all)	0/50	0.64	28.37	18/50	0.93	177,006
Learned ($n_{\min} = 20$, MKNSC only)	0/50	0.64	34.93	16/50	0.92	165,412

Table 5 then shows the results obtained with a node limit and a time limit on the MIPLIB problems. In those experiments, we separated the problems that were solved by all methods from the problems that were not solved by at least one of the compared methods. Similarly to the results obtained on the random problem sets, the proposed branching strategy compares favorably with strong branching both on the node limit and time limit experiments. Nonetheless, the results obtained with the learned branching strategy are still a little below the results obtained with reliability branching. The results presented here are averaged over all considered problems. The detailed results for all problems in the MIPLIB set are available in the supplemental material.

5.3.2. Experiments without limits. Finally, Tables 6 and 7 report the results form our last set of experiments. We apply all branching heuristics on all MIPLIB problems initially contained in Table 3, and let the computers solve the problem for five days[4]. After this time

[4] The computers are given a time budget of five CPU days, after which the B&B process stops automatically. The only exception is the problem instance `pp08aCUTS`, which is solved in due time by all branching heuristics except for

Table 5 Optimization results for the MIPLIB problems of Table 3.

	Solved by all methods			Not solved by at least one method			
Node limit = 10^5 nodes	S/T	Nodes	Time (s)	S/T	Cl. Gap	Nodes	Time (s)
Random	9/44	1,974	2.24	0/44	0.43	10,000	124.50
MIB	9/44	2,532	6.03	6/44	0.50	9,274	233.19
NCB	9/44	879	10.70	11/44	0.72	7,322	232.74
FSB	9/44	692	14.48	12/44	0.73	7,184	629.87
RB	9/44	1,123	15.78	10/44	0.64	7,806	219.39
Learned ($n_{\min} = 20$, all)	9/44	1,194	2.73	10/44	0.62	8,073	162.87
Time limit = 10 min							
Random	19/44	29,588	30.50	0/44	0.47	867,837	600.01
MIB	19/44	14,931	14.68	3/44	0.52	764,439	561.27
NCB	19/44	7,051	41.55	5/44	0.73	101,408	513.00
FSB	19/44	5,687	70.84	3/44	0.66	49,008	534.65
RB	19/44	6,895	27.38	7/44	0.69	257,375	515.40
Learned ($n_{\min} = 20$, all)	19/44	14,008	34.12	5/44	0.63	130,081	512.72

limit, the problems that are not solved by all branching methods are discarded. We create thus a new set of MIPLIB problems (reported in Table 6) that is used to compare different branching strategies until the end of the optimization procedure. Additionally, in another experiment, we let CPLEX use cuts and heuristics (with default CPLEX parameters) in the course of the optimization in order to observe their impact on the efficiency of each branching strategy. The optimization results are shown in Table 7.

Overall, our method compares favorably to its competitors when cuts and heuristics are used by CPLEX. Indeed, in that case, our learned branching strategy is the fastest (almost three times faster than the second fastest method, i.e., MIB) to solve all the 30 considered problems. Note that the apparent bad results of RB are due to three problems that are especially hard for that branching heuristic (air04, air05, and mod011). If we remove them from the computation of the average optimization time, both RB and the learned branching strategy take 74 sec on average to solve the remaining 27 problems. That average time is still 40% smaller than the average time of the runner-up (MIB). These experiments show that our strategy behaves very well when cuts and heuristics are used by CPLEX to solve the problems. The detailed results are available in the supplemental material.

Things appear to be different when cuts and heuristics are not used. Indeed, based on the results of Table 7, our method seems to be very slow, but the large number of nodes

the random branching strategy. We therefore granted additional budget for that configuration (problem + branching heuristic) in order to maximize the number of problems considered in our experiments.

Table 6 Updated list of problems from MIPLIB3 and MIPLIB2003. This list contains the problems from Table 3 that are solved to optimality with each branching heuristic in less than five days.

aflow30a	air03	air04	air05	cap6000	dcmulti
egout	khb05250	l152lav	lseu	mas76	misc03
misc06	misc07	mitre	mod008	mod010	mod011
nw04	p0033	p0201	pk1	pp08aCUTS	qiu
rentacar	rgn	stein27	stein45	vpm1	vpm2

Table 7 Optimization results (until termination) for the updated list of the MIPLIB problems (Table 6).

	w/o cuts and heuristics		w/ cuts and heuristics	
	Nodes	Time (s)	Nodes	Time (s)
Random	7,809,341	29,377.10	152,564	503.38
MIB	3,472,431	7,387.09	105,692	356.52
NCB	145,244	1,136.34	34,500	1,451.74
FSB	129,047	1,597.12	25,941	895.36
RB	318,384	886.12	51,913	2,836.93
Learned ($n_{\min} = 20$, all)	1,037,055	3,023.34	57,652	124.94

and the large amount of time is actually due to a small number of problems for which the method does not work well. These problems artificially increase the average number of nodes and average amount of time reported in the table. A finer analysis can be conducted by interpreting the detailed results reported in the supplemental material. When done, we see that our method is faster than RB in 11/30 cases and faster than FSB in 21/30 cases, thus alleviating the a priori bad performance of the learned branching strategy. A possible explanation for why our method does not perform well on those problems can be that these problems, because too large, are not well represented in the dataset that we use in order to learn the branching strategy. This again shows the importance of considering a large and diverse dataset for the learning of an efficient branching strategy.

6. General observations about the proposed method

The goal of this section is to give some general observations about the method that we propose in this paper, and to try to draw a parallel with other existing branching strategies.

6.1. Comparison with other branching strategies

The idea behind our approach can be seen as a combination of the ideas behind reliability branching and information-based branching. On the one hand, our approach is similar to reliability branching in the sense that we want to find a fast proxy to strong branching. However, the approaches differ in the means by which the information is collected, and in how the information is used. On the other hand, our approach is similar to information-based branching in the sense that it tries to use general information collected during a preliminary phase in order to speed up the current optimization. In both cases, the idea is to obtain a broad overview of the B&B process and to use this overview to take sensible branching decisions. The main difference lies in the way information is collected: information-based branching harvests information through multiple restarts on the same problem, while our method collects optimization information obtained through the optimization of a set of different optimization problems. Another important difference is that, in the case of information-based branching, the collection phase needs to be performed for each problem to optimize, while our approach only requires the information to be collected once. In the light of Table 4, it is to be noted, however, that customizing the branching strategy to the problem being optimized could supposedly yield better results, which would be in favor of the approach that involves collecting some data for each problem to solve.

6.2. Another branching strategy?

The underlying mechanism of our approach is not different from other branching strategies. Indeed, in all cases, features are computed from the current state of the problem (in some way or another) and then used to decide which variable to branch on. Those features can be, e.g., the fractionality of the variable in the current solution (used in most-infeasible and reliability branching), the pseudocosts of the variables (used in reliability branching), or the objective increases observed when the branching is performed (used in strong branching). In our approach, we include many types of features, including some of the aforementioned. The difference lies in how these features are used. While, in traditional methods, the features are assumed to explain everything and are all used to take the branching decision, we let the learning algorithm decide which features are relevant and which are not. The hope is that the machine will make a better choice than the humans when they decide to choose some features and to leave others aside. In that sense, we can see our method as a very general branching strategy that can imitate any other heuristic, as long as the appropriate features are provided.

For example, in its current implementation, our approach can be tied to pseudocost and reliability branching. Indeed, our features include the value of the fractionality of the variables and their pseudocosts (see Section 4). This means that all the information that pseudocost branching uses to take a decision is provided to the learning algorithm. Our approach can thus, in principle, use only the pseudocosts and the variable fractionalities to take a branching decision, if the learning algorithm decides that these features best explain the desired output. But the learning algorithm can also produce a different branching criterion, using other features together with the pseudocosts, hopefully better approximating the desired output. Note that the desired output that we are focusing on is the strong branching score, which is different from the score used by pseudocost branching to take a branching decision.

6.3. Connections with the no free lunch theorems

The *no free lunch* theorems (NFL) (Wolpert and Macready 1997) state that, for certain types of mathematical problems, the performance of any optimization algorithm, averaged over all problems in the class, is equivalent. Additional NFL results indicate that matching, or aligning, algorithms to problems is a way to achieve better performance. This suggests that incorporating knowledge about the problem into the optimization algorithm has the potential to improve its efficiency. Although NFL theorems do not apply to MIP solving and branch-and-bound (Wolpert and Macready 1997), a similar behavior is often observed in practice. For example, while strong branching is very effective (in terms of the number of explored nodes) on general MIP problems, this strategy is not optimal for Constraint Satisfaction Problems (CSP), where other branching strategies are preferred. This indicates that, even if the NFL theorems do not apply as is, it is realistic to imagine that incorporating prior knowledge about the problem could help to improve the performance of traditional MIP solving approaches. The results published in prior work in similar contexts further support this conclusion (see, e.g., Hutter et al. 2010).

Surprisingly, in the MIP optimization area, hybrid branching (Achterberg and Berthold 2009) is the only strategy that takes this fact into consideration. Hybrid branching combines, in a weighted sum, several criteria known to be effective for different types of problems. This probably explains why hybrid branching outperforms other branching strategies across multiple problem sets (Achterberg and Berthold 2009, Achterberg and Wunderling 2013). Despite its simplicity, hybrid branching is a state-of-the-art strategy used in the last CPLEX release (Achterberg and Wunderling 2013). However, one might be interested in what would happen if the criteria were combined in a more elaborated way.

Although our work is completely devoted to the imitation of strong branching, we believe that the same framework can be applied to adapt the branching heuristic to the problem being optimized. In that aspect, the proposed method has a great potential to achieve better performance across large sets of problems, as the branching strategy generated using machine learning can, in principle, imitate decisions made by different branching strategies at the same time.

7. Conclusion

In this paper, we proposed a new approach to design branching strategies for MILP problems. It consists in observing branching decisions taken by a supposedly good strategy, FSB in our case, and to imitate those decisions with a strategy obtained by machine learning. To this end, we develop a set of features that are used to characterize the current state of the problem in the B&B tree from the perspective of a particular variable. This set of features is then used as the input of the learned branching heuristic in order to predict the expected increase of the objective function that a branching on this variable would produce. The experiments show promising results and suggest that further research in this direction may lead to a practical approach to tune branching strategies to particular subclasses of MILP problems.

The underlying mechanism of our approach is not different from other popular branching heuristics. Indeed, in both cases, features are computed from the current state of the problem, and then used to decide which variable to branch on. In our approach, however, we may include many types of features, including those used by popular strategies. The approach is able to sort out which of those features are useful, and to automatically determine how to combine them to rank branching decisions. In that sense, we can see our method as a very general branching strategy that can imitate any other heuristic, as long as the appropriate features are provided. Our method can also discover novel heuristics by combining the features used by several popular methods with novel ones.

Beyond the imitation of existing strategies, another interesting research direction could be to design learning procedures that can distinguish, in a dataset of decisions obtained from different heuristics, the decisions that are productive, from those that are counter-productive.

Acknowledgments

This work was funded by the Dysco IUAP network of the Belgian Science Policy Office and the Pascal2 network of excellence of the EC. AMA's thesis is funded by a FRIA scholarship from the F.R.S.-FNRS. The scientific responsibility rests with the authors.

References

Achterberg, T., T. Berthold. 2009. Hybrid branching. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 309–311.

Achterberg, T., T. Koch, A. Martin. 2005. Branching rules revisited. *Operations Research Letters* **33** 42–54.

Achterberg, T., T. Koch, A. Martin. 2006. MIPLIB 2003. *Operations Research Letters* **34** 361–372.

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

Applegate, D., R.E. Bixby, V. Chvátal, W. Cook. 1995. Finding cuts in the tsp (a preliminary report). Tech. Rep. 05, DIMACS.

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

Berthold, T., D. Salvagnin. 2013. Cloud branching. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 28–43.

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

Breiman, L. 2001. Random forests. *Machine learning* **45** 5–32.

Di Liberto, G., S. Kadioglu, K. Leo, Y. Malitsky. 2013. Dash: Dynamic approach for switching heuristics. *arXiv preprint arXiv:1307.4689* .

Driebeek, N.J. 1966. An algorithm for the solution of mixed integer programming problems. *Management Science* **12** 576–587.

Fischetti, M., M. Monaci. 2012a. Backdoor branching. *INFORMS Journal on Computing* **25** 693–700.

Fischetti, M., M. Monaci. 2012b. Branching on nonchimerical fractionalities. *Operations Research Letters* **40** 159–164.

Geurts, P., D. Ernst, L. Wehenkel. 2006. Extremely randomized trees. *Machine learning* **63** 3–42.

Hutter, F., H. H. Hoos, K. Leyton-Brown. 2010. Automated configuration of mixed integer programming solvers. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 186–202.

Hutter, F., L. Xu, H. H. Hoos, K. Leyton-Brown. 2014. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* **206** 79–111.

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

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

Li, C.M., Anbulagan. 1997. Look-ahead versus look-back for satisfiability problems. *Principles and Practice of Constraint Programming-CP97*. Springer, 341–355.

Patel, J., J.W. Chinneck. 2007. Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Mathematical Programming* **110** 445–474.

Wolpert, D.H., W.G. Macready. 1997. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on* **1** 67–82.