ON LEARNING AND BRANCHING: A SURVEY

Andrea Lodi
Giulia Zarpellon

April 2017

DS4DM-2017-004

POLYTECHNIQUE MONTRÉAL
DÉPARTEMENT DE MATHÉMATIQUES ET GÉNIE INDUSTRIEL
Pavillon André-Aisenstadt
Succursale Centre-Ville C.P. 6079
Montréal - Québec
H3C 3A7 - Canada
Téléphone: 514-340-5121 # 3314

Noname manuscript No.
(will be inserted by the editor)

On learning and branching: a survey

Andrea Lodi · **Giulia Zarpellon**

Received: April 6, 2017/ Accepted: date

Abstract This paper surveys learning techniques to deal with the two most crucial decisions in the branch-and-bound algorithm for Mixed-Integer Linear Programming, namely variable and node selections. Because of the lack of deep mathematical understanding on those decisions, the classical and vast literature in the field is inherently based on computational studies and heuristic, often problem-specific, strategies. We will both interpret some of those early contributions in the light of modern (Machine) Learning techniques, and give the details of the recent algorithms that instead explicitly incorporate Machine Learning paradigms.

Keywords Branch and Bound · Machine Learning

1 Introduction

In the last decade we have experienced the impressive development of powerful artificial intelligence algorithms able to perform complex task in the form of so-called "predictions" in contexts as diverse as image recognition, natural language interpretation, word alignment, etc. Those algorithms are not only theoretical but, in addition, they have been effectively deployed into reliable software packages commonly used by all sort of intelligent devices, computers, sensors, smart TVs and smart phones. This revolution, whose potential is not yet fully understood and not yet fully realized, has been possible because of two main ingredients. On the one side, the impressive increase of computing power (especially here Graphical Power Units) paired with the

A. Lodi
Canada Excellence Research Chair, École Polytechnique de Montréal
C.P. 6079, Succ. Centre-ville, Montréal, Québec, Canada H3C 3A7
Tel.: +1-514-3404711
Fax: +1-514-3404463
E-mail: andrea.lodi@polymtl.ca

G. Zarpellon
E-mail: giulia.zarpellon@polymtl.ca

enormously extended technological (hardware and software) capability of collecting data, often in the form of examples. On the other side, the shift by (part of) the Artificial Intelligence community, that of Machine Learning (ML, in short), of the learning paradigm from "knowledge formalization" to "learning by examples", which enables perception and a form of learned intuition.

Of course, that stream of success has attracted the attention not only of the business world but also of other scientific communities that became interested in exploring the possibility of using ML techniques within their algorithms and methods, to be able to tackle structural challenges that have resisted traditional approaches. Clearly, this applies to the context of using ML algorithms within domain applications as healthcare, transportation, energy, and virtually anywhere a knowledge acquisition is required by the decision-making process.

However, for mathematical optimization, the most fascinating question concerns the use of ML techniques in the algorithmic design, independently of the application domain in which optimization algorithms are used. This question can be asked in many contexts and it is certainly related to the fact that breaking ties in optimization algorithms is far from perfect, see, e.g., Koch et al (2011) and Lodi and Tramontani (2013). Indeed, learning mechanisms able to discover structural properties of seemingly equivalent components of an algorithm would certainly be very useful. For example, given the *Mixed-Integer Linear Programming problem* (MILP)

$$\min\{c^T x : Ax \geq b, x \geq 0,\ x_i \in \mathbb{Z}\ \forall i \in I\}, \tag{1}$$

learning a "better" initial basis among the equivalent (optimal) ones of the *linear programming* (LP) relaxation

$$\min\{c^T x : Ax \geq b, x \geq 0\}, \tag{2}$$

could lead to a reduction of the so-called *performance variability* (see again Lodi and Tramontani 2013) as well as be beneficial from the performance standpoint (see, e.g., Fischetti et al 2016).

In this survey, we concentrate on the questions, within such a flavor that we can call *learning for optimization*, that can be asked concerning the crucial decisions of the most well-known exact method for discrete optimization, i.e., the *branch-and-bound* algorithm (Land and Doig 1960).

In its basic version the branch-and-bound algorithm iteratively partitions the solution space into sub-MILPs (the children nodes) which have the same theoretical complexity of the originating MILP (the father node, or the root node if it is the initial MILP). Usually, for MILP solvers the branching creates two children by using the rounding of the solution of the LP relaxation value of a fractional variable, say x_ℓ, constrained to be integral, $\ell \in I$,

$$x_\ell \leq \lfloor x^*_\ell \rfloor \quad \vee \quad x_\ell \geq \lceil x^*_\ell \rceil, \tag{3}$$

where x^* denotes the optimal solution of (2). The two children above are often referred to as *left* (or "down") branch and *right* (or "up") branch, respectively. On each of the sub-MILPs the integrality requirement on the variables $x_i, \forall i \in I$ is relaxed and the LP relaxation is solved. Despite the theoretical complexity, the sub-MILPs

become smaller and smaller due to the partition mechanism (basically, some of the decisions are taken) and eventually the LP relaxation is directly integral for all the variables in I. In addition, the LP relaxation is solved at every node to decide if the node itself is worthwhile to be further partitioned: if the LP relaxation value is already not smaller than the best feasible solution encountered so far, called *incumbent*, the node can safely be fathomed because none of its children will yield a better solution than the incumbent. Finally, a node is also fathomed if its LP relaxation is infeasible.

Paired with the *cutting plane* algorithm (Gomory 1960) to obtain variations of the *branch-and-cut* paradigm (Padberg and Rinaldi 1991), the branch and bound is the most basic structural component of modern MILP solvers (see, e.g., Lodi 2010; Linderoth and Lodi 2010). As pointed out by Lodi (2013), the *exact* computation performed by MILP solvers relies on a somehow surprising collection of *heuristic* decisions, two of the most crucial ones being those associated with the branching scheme outlined above, namely

- *variable selection*, which of the variables x_ℓ, $\ell \in I$ among those fractional at any node, to branch on in (3), and
- *node selection*, which of the currently open nodes to process next.

In Section 3 the most traditional and effective strategies to deal with the two decisions above are discussed. This survey documents the recent attempts to incorporate sophisticated learning mechanisms within those strategies. In order to do that, we present in Section 2 a brief overview of the machine learning concepts that are required to understand the algorithms surveyed in Sections 4 and 5. Finally, Section 6 discusses a few additional references related to learning within a branching tree and draws some conclusions outlining some possible research directions.

2 A brief overview of Machine Learning

As already mentioned in Section 1, machine learning is the subfield of artificial intelligence devoted to develop intelligent systems that learn from experience (i.e., from examples, or observations) how to perform a given task. In ML, the prediction process is performed in an operational way, using information coming from data and following some specified criterion; optimization is undoubtedly one of the cores of this process.

The aim of this section is to make the reader familiar with the essential concepts of learning: we will introduce the traditional learning framework and its standard tasks of supervised and unsupervised learning in Section 2.1. We will then mention in Section 2.2 some issues and pitfalls of learning that every researcher using ML should be aware of while developing an application, and we will conclude the section with a brief introduction to another learning paradigm called reinforcement learning in Section 2.3.

For more details and material we refer to Bishop (2006), Hastie, Tibshirani, and Friedman (2009), Goodfellow, Bengio, and Courville (2016) and Sutton and Barto (1998).

2.1 The standard learning setting and tasks

We can formalize the traditional learning framework starting with a set of n examples $\mathcal{D}_n = \{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_n\}$, where we denote with $\mathbf{z}_i$ the realization of a random variable Z_i under an unknown process $P(Z)$. Each $\mathbf{z}_i$ is supposed to be independently drawn from the distribution specified by $P(Z)$.

One wishes to learn a function f exploiting some characteristics of $\mathcal{D}_n$ (and hence $P(Z)$), with the aim of employing it to make future predictions on new examples. The search for f is performed among the members of a certain (parametrized) family of functions $\mathcal{F}$. Together with $\mathcal{F}$, a loss functional $\mathcal{L} : (\mathcal{F}, \mathcal{D}_n) \to \mathbb{R}$ is specified in order to evaluate the quality of different available $f \in \mathcal{F}$.

The ultimate goal would be to find f^* as an optimal minimizer of $\mathbb{E}[\mathcal{L}(f,Z)]$, the expected value of $\mathcal{L}(f,Z)$ under $P(Z)$, following the so-called *expected risk minimization* principle. However, being compelled to work in a restricted space of functions $\mathcal{F}$ and not knowing $P(Z)$ *a priori*, one aims instead at minimizing the *empirical risk*, using the examples of the finite set $\mathcal{D}_n$ to learn a function

$$f_{\mathcal{D}_n} \in \operatorname{argmin}_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f, \mathbf{z}_i), \tag{4}$$

which will then be employed as a predictor.

The examples $\mathbf{z} \in \mathcal{D}_n$, the loss functional $\mathcal{L}$ and the prediction goal of the to-be-learned function $f \in \mathcal{F}$ can be various: such differences in forms and intents are what characterize different ML tasks. Although not in the attempt of being exhaustive, we will briefly go through two main types of learning: supervised and unsupervised ones. The intent is to familiarize with the general framework, in order to properly locate the algorithms that will be considered later on.

Supervised learning. In supervised learning, data in $\mathcal{D}_n$ consist of pairs $\mathbf{z} = (\mathbf{x}, \mathbf{y})$; we call $\mathbf{x} \in \mathbb{R}^d$ the *input* or *features* vector, and $\mathbf{y}$ the *output* or *target*. Depending on the output type, two main predictive tasks can be identified:

- ◇ Classification: the target is a qualitative label, used to differentiate between $m \in \mathbb{N}$ classes or categories. The scalar label $y \in \{1, \dots, m\} \subset \mathbb{Z}$ can be encoded in $\mathbf{y} \in \mathbb{Z}^m$, and we could for example choose to measure the accuracy of a classifier f by means of the classification *error rate*: often, the *expected 0-1 loss* is evaluated by considering the loss of an example as $\mathcal{L}(f, (\mathbf{x}, \mathbf{y})) = I_{\{f(\mathbf{x}) \neq \mathbf{y}\}}$, where $I_{\{\cdot\}}$ denotes the indicator function.
- ◇ Regression: the target is a quantitative output $\mathbf{y} \in \mathbb{R}^m$; the prediction $f(\mathbf{x}) \in \mathbb{R}^m$ of a regressor f estimates the expected value of $\mathbf{y}$ given $\mathbf{x}$. An example of loss functional commonly used in this setting is the *quadratic error* $\mathcal{L}(f, (\mathbf{x}, \mathbf{y})) = \|f(\mathbf{x}) - \mathbf{y}\|^2$.

Unsupervised learning. As the name suggests, in the paradigm of unsupervised learning the prediction is performed without a "supervisor" knowing the correct answers: data does not come with a specified target, but only as input of features $\mathbf{z} = \mathbf{x} \in \mathbb{R}^d$. The aim is to learn a function f describing in some way the unknown process $P(Z)$ from which the examples were drawn. Some common tasks in this setting are:

- ◇ Density estimation: f is an estimator of the distribution of input data. Since the concept of error rate is not suited for this unlabeled context, one could aim at maximizing the *(log-)likelihood* of the observed data, i.e., their probability with respect to the underlying distribution $P(Z)$.
- ◇ Clustering: the purpose of the learning is to discover similarities within the input space. Data can be grouped in hard-cut clusters or within a soft partition of the space; for a new point we predict its memberships to one or several groups.
- ◇ Dimensionality reduction: a new representation of the input data is constructed, usually in a lower dimensional sub-space of the original input space (aiming at data visualization, for example). The goal is to identify some important characteristics of the input $\mathbf{x}$, e.g., via selection or extraction.

2.2 Few things to keep in mind

Undertaking a project involving ML can be a non-smooth path to pursue, especially for beginner practitioners with a non ML-related background. The purpose of this subsection is to warn the reader against some non-trivial issues that could easily be encountered along the way.

The extensive discussion of those issues and the methods to overcome them is beyond the scope of this paper. Instead, we point out the importance of performing learning with awareness, i.e., by following the best practices of the field. An interesting outlook of these and other concerns can be found in Domingos (2012).

Features engineering. A key step in every ML application is the design of what the input data represent, i.e., what are the shape and the meaning of the examples $\mathbf{z}$ that the learned function $f_{\mathcal{D}_n}$ should be receiving as argument in order to make a prediction. Often, one would pre-process raw data $\mathbf{x}$ into $\phi(\mathbf{x})$ by means of a features extraction procedure, in order to find a better suited representation of the problem, to be fed to a learning algorithm with. With a minor abuse of notation, we will treat input and features indifferently, denoting them as $\mathbf{x}$ or $\phi(\mathbf{x})$ or ϕ, depending on the context.

The features vector $\mathbf{x}$ encoded in the sample $\mathbf{z}$ should represent the problem at hand accurately: features are domain-specific and require some *a priori* knowledge about the nature of the problem, thus they are most often manually designed. Systems like those used in *Deep Learning* (Goodfellow et al 2016) are able to learn valuable features automatically, but the human intervention is still needed in the task of tuning the resulting complex architecture. In learning, it is possible for a single feature to be irrelevant if considered alone, and to become very significant when combined with other traits.

While the intuition suggests that the more features we have, the more information we will gain and the better the prediction will be, things are not as trivial. The risk is to incur in the so-called *curse of dimensionality* (Bellman 1961): having many features corresponds to working in a high-dimensional space, where a limited dataset could result in a very sparse sampling, and the locally-based assumption that similar examples lead to similar predictions might fail.

Anyhow, features engineering is an aspect of learning that requires care and an iterative trial process, during which some features could be added, some discarded, some others combined, without a precise recipe to follow.

Generalization, overfitting and model selection. The ability to perform *generalization* is the fundamental property to look for in a learned predictor. However, the fact that we are bind to search within $\mathcal{F}$, and the availability in $\mathcal{D}_n$ of only a finite number of observations, make the learning of an optimal predictor a delicate task. One of the most frequent trap when dealing with learning is *overfitting*.

Broadly speaking, the phenomenon of overfitting represents the inability of a learned function to generalize: this occurs because $f_{\mathcal{D}_n}$ as in (4) is optimized to fit the specific dataset $\mathcal{D}_n$, making $\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f_{\mathcal{D}_n},\mathbf{z}_i)$ an underestimate of the expected risk (also called *generalization error*), which is computed on new never-seen examples. In particular, the more the learned function fits the data in $\mathcal{D}_n$, the more the generalization error will be optimistic. Overfitting is strictly related to the choice of *model complexity* and the so-called *bias-variance trade-off*, and makes it necessary to use some caution when establishing the "best" learning model among many.

In order to compare the performances of different learning models, a proper model selection procedure prescribes to divide the dataset into three parts, to be used in distinct *training*, *validation* and *test* phases. Different hyper-parameters versions of a chosen model are considered; each version's parameters are optimized during the training phase on the first share of the dataset. The learned models are then ranked with respect to their predicting performance on the validation set, and the hyperparameters setting resulting in the best performance is selected. Finally, to determine the generalization performance of the ultimately chosen model, a neutral test set is employed; examples that were never faced before are used in order to avoid a biased, optimistic estimation.

For further details on overfitting, standard procedures to avoid it and additional references, we refer to Hastie, Tibshirani, and Friedman (2009), ch. 7.

2.3 Another paradigm: reinforcement learning

We introduce now a third type of learning scheme, deviating from the traditional setting we outlined in Section 2.1. *Reinforcement learning* is concerned with how an agent learns while interacting with an uncertain environment in order to maximize its reward. The agent is able to perceive some information about the state of the system it lives in, and can take state-changing actions that result into a *reward signal* (or a punishment one), evaluating its behavior. Each action affects later inputs of the system, and hence subsequent rewards; the goal is that of finding a policy maximizing the long-term return.

The setting of a Markov decision process to be optimally controlled provides a (partial) theoretical framework for reinforcement learning. At discrete time steps $t = 0, 1, 2, \ldots$, the agent observes the environment state $s_t \in \mathcal{S}$, and subsequently takes an action $a_t \in \mathcal{A}(s_t)$ with probability $\pi(a_t|s_t)$. The state changes into s_{t+1} with probability $P^{a_t}_{s_t,s_{t+1}}$, and an immediate reward $R^{a_t}_{s_t,s_{t+1}}$ is received. The goal is to learn an

optimal policy $\pi : \mathcal{S} \to \mathcal{A}$ typically maximizing the expected long-term *discounted return*

$$G(s_0, \pi) = \mathbb{E}[R^{a_0}_{s_0,s_1} + \gamma R^{a_1}_{s_1,s_2} + \cdots + \gamma^t R^{a_t}_{s_t,s_{t+1}} + \ldots], \tag{5}$$

where $\gamma \in [0,1)$ is the discount rate. The concept of discounting is useful in this framework to model the fact that a future reward is worth less than an immediate one. In particular, if $\gamma = 0$, the impact of future actions is not taken into account, i.e., the agent is short sighted and maximizes at each step the immediate reward only, potentially reducing its total gain. Usually, $G(s_0, \pi)$ is referred to as the *value* of state s_0 following the policy π.

A classical reference for reinforcement learning is Sutton and Barto (1998); additional material can be found in Bertsekas and Tsitsiklis (1996) and Szepesvari (2010).

Note that this third paradigm is inherently different from the previous two: while in supervised learning we are provided with examples correctly labeled with their "optimal" response, in reinforcement learning an agent has to learn from its own experience what a good behavior is, by a process of trial and error. Moreover, though an agent indeed discovers some implicit characteristics of its environment, the ultimate goal of reinforcement learning is that of maximizing a reward, and not that of finding hidden structures within the unlabeled examples, as it is instead for unsupervised learning.

A peculiar concern of reinforcement learning is the trade-off between *exploration* and *exploitation*. On the one hand, the agent should explore the space by trying new actions to know if they are valuable, whereas, on the other hand, it should exploit those actions yielding a high reward that it has already experienced. Any kind of unbalanced learning will produce poor results.

3 The branch-and-bound framework

As anticipated in Section 1, the variable and node selections are largely seen as the most crucial decisions in exact methods for MILP. On the one hand, branching on a variable that does not lead to any serious simplifications on any of the (two) children can be seen as doubling the size of the tree with no improvement, thus leading to extremely large (out of control) search trees. On the other hand, effective MILP solvers need to provide a good compromise between finding good solutions quickly and the chance of proving optimality in the short-to-medium term, a trade-off which is of course related to the way the search tree is explored.

Variable selection problem. This is the task of deciding how to partition the current node, i.e., on which variable to branch on in order to create the two children. For a long time, a classical choice has been branching on the most fractional variable, i.e., in the 0-1 case the closest to 0.5 (sometimes referred to as *most infeasible branching*, MIB in short). That rule has been computationally shown by Achterberg et al (2005) to be worse than a complete random choice. However, it is of course very easy to evaluate. In order to devise stronger criteria one has to do much more work. The extreme is the so called *strong branching* (SB, in short) technique (see, e.g., Applegate

et al 2007; Linderoth and Savelsbergh 1999). In its full version (FSB, in short), at any node one has to tentatively branch on each candidate fractional variable and select the one on which the increase in the bound on the left branch times the one on the right branch is the maximum. Of course, this is generally unpractical but its computational effort can be easily limited in two ways: on the one side, one can define a much smaller candidate set of variables to branch on and, on the other hand, can limit to a fixed (small) amount the number of Simplex pivots to be performed in the variable evaluation. Another sophisticated technique is *pseudocost branching* (PC, in short) which goes back to Benichou et al (1971) and keeps a history of the success (in terms of the change in the LP relaxation value) of the branchings already performed on each variable as an indication of the quality of the variable itself. Among the most recent effective and sophisticated methods, *reliability branching* (RB, in short) (Achterberg et al 2005) integrates strong and pseudocost branchings. The idea is to define a reliability threshold, i.e., a level below which the information of the pseudocosts is not considered accurate enough and some strong branching is performed. Such a threshold is mainly associated with the number of previous branching decisions that involved the variable. Finally, *hybrid branching* (Achterberg and Berthold 2009) computes for each candidate variable five different measures, chosen among typical branching scores of MILP, constraint satisfaction and satisfiability technologies. The measures are first normalized and then combined into a single score by means of a weighted sum.

Node selection problem. This is the most classical task of deciding how to explore the tree: one extreme is the so called *best-first* strategy in which one always considers the most promising node, i.e., the one with the smallest LP value, while the other extreme is *depth-first* where one goes deeper and deeper in the tree and starts backtracking only once a node is fathomed, i.e., it is either (mixed-)integer feasible, or LP infeasible or it has a lower bound not better (smaller) than the incumbent. The pros and cons of each strategy are well known: the former explores less nodes but generally maintains a larger tree in terms of memory, while the latter can explode in terms of nodes and it can, in the case some bad decisions are taken at the beginning, explore useless portions of the tree itself. All other techniques, more or less sophisticated, are basically hybrids around these two ideas, like interleaving best-first and diving, i.e., a sequence of branchings without backtracking, in an appropriate way. Some of those techniques are discussed in Section 5.1.

Besides the fact that the above decisions are highly crucial for the effectiveness of the branch-and-bound (B&B, in short) framework and, in general, for the MILP technology, the urge of trying to use sophisticated learning mechanisms to guide them is motivated by their poor understanding from the mathematical standpoint. In other words, there is no deep understanding of the theory underneath branching, if any exists, so the application of (modern) statistical methods seems quite appealing. The next two sections defines the core of the present survey by systematically introducing ML approaches to variable selection, Section 4, and node selection, Section 5.

4 Variable branching heuristic

In general terms, all branching rules aim at guiding the search of the B&B tree in an efficient way, by appropriately choosing at each node the fractional variable one ought to branch on. At every step of the search, plenty of heterogeneous information can be gained, and a meaningful branching strategy should integrate and exploit this by updating knowledge in its decision-making rule.

The common denominator of the papers discussed in this section is their attempt of defining of a branching strategy extracting novel kinds of information and combining them in original ways. In the most recent works, the aggregation of the collected information is performed by means of ML algorithms. Nonetheless, we present as well some early works not mentioning the ML framework, which we could consider precursors in conveying the idea of gathering more and diverse data (in a ML framework, one would call them features) to capture the state of the B&B system and improve its decision-making process.

As observed by Marcos Alvarez (2016), this idea of extracting some characteristics to derive a branching rule is indeed what traditional heuristic schemes for branching already perform: fractionalities are employed in most-fractional branching; LP gains, measuring the impact that a candidate variable could have on the objective function value, are computed in SB and estimated with pseudocosts, i.e., taking into account historical data of the search.

In the recent works we will discuss, the novel trait is that of exploiting (possibly a large quantity of) collected data, and employing the learning framework to come up with more informed and complex decision functions, estimating a good branching strategy.

With the underlying belief that a more sophisticated and high-performing branching rule could be detected, the following works explore different pertaining questions: Which information should be used? How could it be efficiently extracted and appropriately learned? Which criteria should guide the search?

The section opening is devoted to the discussion of some "forerunner" works (see Section 4.1). We will see how they already implied shared views on the role of learning for branching, on which recent ML-based attempts (treated in Section 4.2) are currently building up.

4.1 Precursors of "learning-to-branch"

The possibility of improving the general B&B performance by means of collecting and exploiting more information than customary is already questioned in Glankwamdee and Linderoth (2011). The authors investigate whether typical B&B information extracted *two* levels deeper than a given node would influence the decision of the branching variable. The devised *lookahead branching* rule aims at maximizing both bound improvement and node pruning, and the gathered additional information proves to be useful not only for the purpose of defining a new branching rule, but also for auxiliary tasks such as bound fixing and simple implications deduction.

Clearly, the computational effort of performing such a forward scouting is significant. Hence, an abbreviated (and cheaper) version of the scheme is also defined. Even though the total number of explored nodes is reduced in certain instances if compared with a classical one-step lookahead SB implementation, the authors themselves point out that lookahead branching is too costly to be likely employed as a default branching scheme for MILP. However, it might be that for some classes of problems one might be willing to afford some supplementary computational cost, if the effort could somehow pay off.

As many other traditional heuristics, lookahead branching relies on LP gains to measure the impact of a candidate variable with respect to the collected additional information. A different point of view is developed in Karzan, Nemhauser, and Savelsbergh (2009), where the measure of impact is based instead on fathoming decisions. This choice is motivated by two broadly acknowledged assumptions:

(i) the final goal in branching is minimizing the total number of explored nodes; in this sense, a *node-efficient* method is sought;
(ii) branching decisions are more crucial at the top of the tree.

The paper investigates the idea of a three-phase method for binary MILP problems. Specifically, a *clause* is defined as a partial assignment of the binary variables that cannot lead to possible improvements of the objective function. The authors exploit the fact that in any binary B&B tree a fathomed node gives a clause, and that this kind of information can be further strengthened to yield more fathoming. First, a given instance is partially solved: within an upfront *collection phase*, clauses associated with fathomed nodes of the incomplete B&B tree are gathered, until their number reaches a threshold (fixed at 200). In the subsequent *improvement phase* the basic information is refined by solving an auxiliary MILP problem, in order to be finally employed in the *restart phase*, when the instance is fully solved with the gained information.

Some fathoming-based branching rules are defined, taking into account various possibilities for assigning weights to the collected clauses and estimating the effect of fixing and branching on a candidate variable, in a fashion that reminds combination rules for pseudocosts (see, e.g., Linderoth and Savelsbergh 1999). Moreover, the improved information is exploited for two additional tasks. The combination of fathomed-based branching with *cuts generation* and a sort of *propagation* yield improved performance with respect to CPLEX (2017) (version 11.1) with and without dynamic search.

However, there is no clear winner among the tested strategies, and it is not obvious to identify the benefits of using such improved information for branching. Indeed, more information associated with clauses could potentially result in longer computing times. Having in mind a ML framework, we could think of the proposed collection phase as a kind of training phase personalized (and repeated) for each instance, where the clause information themselves (we could call them features) are instance-specific.

Partially following the work of Karzan et al (2009) is the *backdoor branching* approach of Fischetti and Monaci (2012a). In this work, the collected data include some

fractional solutions that are characterized as hurdles for the LP gap reduction. The learning phase is in fact a *sampling phase*, consisting of a multiple restart scheme. Iteratively, a given problem is partly solved until a certain number $K_{max}(=10)$ of fractional solutions are encountered. The collected fractionalities become the input of a set covering model, computing a minimum cardinality *backdoor*, i.e., a minimum set S of branching variables whose integrality is enough to ensure that a certified optimal solution value is reached. Variables in S are assigned a priority to be chosen for branching, and a new updated incomplete run is performed. After $R(=10)$ iterations, or when a backdoor S with $|S| > \Gamma(=5)$ is found, a final *long run* is executed, employing a MILP solver as a black-box to which only the ultimate priorities are specified.

The method is compared with two settings of CPLEX (version 12.2): the default solver (with no cutoff provided), and a variant sharing the same setting as the tested algorithm, in which the optimal solution is provided, and cuts, heuristics and variable aggregation in preprocessing are deactivated. Backdoor branching compares well with the competitor method sharing its similar setting, and turns out to be very helpful in expanding the top levels of the tree effectively.

Note that the purpose of the designed *backdoor branching* procedure is not that of selecting a single variable by means of a score. Instead, the goal consists of identifying a subset of variables that are in some sense top-ranked with respect to a certain branching priority measure; a similar idea will be encountered later on. The authors briefly mention the trade-off between collecting (more) reliable information and the cost coming with it. Two nice-to-have properties of prospective features can clearly be outlined.

- ◇ Features should be *relevant*, i.e., they ought to precisely and (as far as possible) completely describe those aspects of the B&B system playing a key role in the optimization and its efficiency.
- ◇ Features should be *low-cost* to compute, i.e., they should not constitute a computational burden.

The latter property could actually be drawn from a more general assumption, legitimate in the definition of an effective branching:

(iii) while being node-efficient, a good branching scheme is *time-efficient* as well.

Note that in backdoor branching, as in Karzan et al (2009), the learning phase is actually a sampling phase that needs to be repeated for each instance, thus not qualifying as a learning-to-generalize mechanism. For each problem, shallow explorations of the search tree are performed, in order to figure out a likely good path in the final run. Finally, observe that in both Karzan et al (2009) and Fischetti and Monaci (2012a) the collected information is manipulated by (optimally) solving MILP or LP problems, somehow learning by relying only on MILP technology.

A completely different paradigm is explored in Gilpin and Sandholm (2011), where a typical AI tool such as Information Theory (Shannon 1948) is used to derive branching rules. The approach is motivated by the interpretation of the B&B tree evolution

as a search process, carrying certain and uncertain information. The essential observation is that nodes at the beginning of the search hold a high amount of uncertainty about the values of the variables, while at the end of the tree there is no uncertainty of this kind left. The idea is hence to guide the search in order to remove uncertainty, or, in other words, to propagate as much as possible the sure information carried by variables. In practice, the fractional values of integer-constrained variables, i.e., the basic feature of the most-fractional branching rule, are treated as probabilities, indicating the confidence in expecting the variable to be greater than or equal to its current value in the optimal solution. To measure the amount of uncertainty / information of a variable, the notion of *entropy* (Shannon 1948) is employed.

Four diverse families of *entropy branching* (EB, in short) heuristics are designed. The first family performs branching with one-step lookahead as in SB but choosing the variable yielding children with smallest uncertainty. Note that EB alone does not employ the objective function in taking a branching decision. The second family explores hybrid approaches: SB and EB scores are combined in terms of pure ranking positions or by means of a weighted sum. Additionally, EB is tested as tie-breaker for the classical SB rule. A third family includes methods that do not perform lookahead, but use instead the LP values of a current solution; two rules are defined in the special contexts of combinatorial procurement auctions and facility location problems (see again Gilpin and Sandholm 2011). The last proposed family deals with multi-variables branches, considering branching on the sum (of values) of a subset of variables. The purpose is that of selecting the set of variable to be branched on as that resulting in the smallest entropy in a one-step lookahead.

Computational experiments performed on MIPLIB 3.0 (Bixby et al 1996) show no clear winner between SB and EB for the defined rules, while EB outperforms SB on some hard real-world procurement instances. Apart from the computational results, the authors' high-level discussion on branching approaches promotes further thinking about the various kinds of information that should be employed in a good branching strategy. In their point of view, different strategies refer to the different ways one could try to reach the goal of node-efficiency in B&B. Since a path in the tree can end in three possible ways, see Section 1, the authors try to interpret the existing branching methods with respect to the concurrent goals of driving the search towards early fathoming, early feasibility and early integrality.

We conclude the overview of forerunner works with Liberto, Kadioglu, Leo, and Malitsky (2016), where a more framed use of ML techniques for variable branching heuristics is implemented. Slightly detaching itself from the others, this work does not aim at finding a new branching rule, but instead at best combining some existing ones along the tree. The motivating background is that of *portfolio algorithms*, where given a set of different methods one wants to predict and use the best available method with respect to the instance to be solved. Such techniques are based on the observation that it is unlikely that it does exist a single method dominating all others for every instance, and hence one looks for a dynamic method able to behave with flexibility, in an instance-specific way.

The idea of Liberto et al (2016) is to devise an algorithm dynamically switching between different branching heuristics along the branching tree, choosing among

them with respect to the different encountered subproblems. The goal of *DASH* (*Dynamic Approach for Switching Heuristics*) is precisely that of guiding the search by means of selecting the best branching rule, following the changing state of the B&B system. A similar dynamic scheme was introduced in Kadioglu et al (2012) for the special case of set partitioning problems.

In Liberto et al (2016), the authors define a features space comprising 40 traits that capture aspects of the subproblem regarding its MILP formulation as well as its position in the B&B tree. As sought, features computation does not constitute an expensive overhead. A portfolio of six traditional branching heuristics is implemented: most(less) fractional rounding, most(less) fractional and highest objective rounding, pseudocosts branching with weighted score and product score. Finally, a dataset of 341 instances coming from heterogeneous benchmark sets is considered, and split into a training and a test set. The first learning step is a clustering of problems motivated by the assumption according to which problems with similar features will yield to the same chosen heuristic, . The grouping of instances is carried out by the *g-means* algorithm of Hamerly and Elkan (2003), a method similar to the classical *k-means* of MacQueen (1967), which additionally determines in an automatic way the optimal number of clusters, assuming an underlying Gaussian density distribution. In particular, an extended training set is provided for clustering, in which some computed subproblems of the original training instances are added, with the intent of making the dataset more representative of the subproblems in the tree. Once the clusters are identified, a "best" heuristic is assigned to each of them. Clearly, this assignment is delicate and a key component of the entire procedure: given the continuous changes of the subproblems types along the branching tree (partly due to the chosen branching heuristic itself!), the assignment need to be performed simultaneously for all clusters. The step is undertaken by the parameter tuner GGA (Ansótegui et al 2009), and only the original (i.e., not extended) training set is employed.

Once the training setup is completed, at a given node of the tree, DASH computes the features of the subproblem and its nearer cluster (the Euclidean distance with respect to the clusters' centers is measured), and subsequently employs the assigned "best" branching heuristic for the selected cluster. In practice, switches do not happen at every node. This enforced limitation is motivated by the wish of further reducing the computational cost of features extraction, and by the fact that features change progressively along the tree, as it is shown by a 2-dimensional *Principal Component Analysis (PCA)* (Abdi and Williams 2010). As a consequence, the switch is activated only up to the 10th depth-level of the tree and just at some prefixed points in time (every 3rd node); in all other cases, the parent node's heuristic is maintained as the default choice.

Experimentally, DASH compares well against static and randomly switching heuristics counterparts. The authors present a pair of variants (DASH+ and DASH+filt), allowing the choice of not switching heuristic inside a defined cluster, and performing a feature selection operation as well. The higher the degree of flexibility and information selection of the algorithm, the better the numerical results seem to be.

Some supplementary remarks. In Liberto et al (2016) as in Gilpin and Sandholm (2011), another characteristic of an ideal branching heuristic emerges. Namely,

(iv) given the highly dynamic and sequential nature of B&B, a branching scheme should be *adaptive*, not only with respect to different instances, but with respect to the whole tree evolution as well.

We observe that Liberto et al (2016) constitutes an initial attempt of using the idea of exploiting a larger and more complete set of problems information by typical ML tasks such as clustering and dimensionality reduction. However, the method faces some limitations. First, the offline and rigid clustering upfront seems to clash with the dynamic and ever changing nature of B&B subproblems. Note that the identified need of considering the evolution of the tree, as expressed in (iv), needs to be balanced with (iii) time-efficiency, resulting in the depth and intervals prescriptions. Finally, the role played by different portions of the dataset within the various learning steps is not clear. A precise procedure of model selection should be performed, in order to avoid hidden overfitting and other pathological behaviors.

Before we move on to the next section, we summarize the recognized properties a learned branching rule ought to incorporate:

(i) node-efficiency,
(ii) focus on top-levels of the tree,
(iii) time-efficiency,
(iv) adaptiveness within the tree evolution.

We are going to see how similar considerations are addressed in the following few works, exploiting more closely the ML framework.

4.2 Novel ML-based branching heuristics

With the exception of DASH (Liberto et al 2016), all other works that were discussed up to now aim at defining a new branching strategy by means of using various types of information and originally assemble branching decision rules. We will present in this section some attempts in taking further the "precursors" underlying ideas, having as goal that of building learned branching schemes. The novelty of these approaches resides in their more methodological use of the ML framework, as we presented it in Section 2.

Reliability branching and information-based branching (Karzan et al 2009) inspired the work of Marcos Alvarez, Louveaux, and Wehenkel (2017). Given the fact that SB-like decisions are considered good decisions for branching, in that they minimize the number of explored nodes, but coming with a very high computational cost, the idea of approximating and speeding up SB has been already explored by methods such as RB and the *non-chimerical branching* (NCB, in short) of Fischetti and Monaci (2012b). In Marcos Alvarez et al (2017), the aim is that of learning one efficient approximation of SB by means of supervised learning techniques. The proposed method consists of two main phases. First, features are extracted to depict the state of a candidate branching variable within a specific node of the tree. The full SB decisions are recorded by solving to optimality a set of training instances, and a regressor

is learned to predict estimated SB scores. After that, the learned heuristic is employed as branching heuristic for future B&B runs.

Before going further into details, it is worth to remark few things. Note that in the first part of the approach SB is still (heavily) employed, before the switch to the learned heuristic takes place. This reminds of procedures such as *hybrid strong / pseudo- cost branching* (SB+PC, in short) where SB is employed only until a certain depth, and then switched for PC (Achterberg et al 2005), and reliability branching.[1] Indeed, the use of supervised learning calls for labels, meaning that we still need good SB scores as examples from which to learn. Moreover, B&B trees are now fully explored in the training phase: the exploration is deeper than those proposed in Fischetti and Monaci (2012a) and Karzan et al (2009), which focus instead on the top levels only. However, with respect to these two works, the expensive training phase is now performed once and for all. The offline upfront aims in fact at generalizing a prediction across all possible future instances, and hence does not need to be repeated for each of them.

Within the features design process of Marcos Alvarez et al (2017), the trade-off between relevance and expense is treated with care. Moreover, the authors identify other three desirable properties that a set of features for branching should include:

- ◇ *size-independence*, if one aims at learning a function able to generalize across instances of various size;
- ◇ *invariance* with respect to irrelevant changes within the instance, e.g., row or column permutations;
- ◇ *scale-independence*, meaning that features should not change if parameters c, A and b (as in (1)) are multiplied by some factor.

The defined features are divided into three main groups.

- ◇ *Static problem features* describe the role of candidate variable x_i with respect to the problem's parameters c, A and b, and are computed only once.
- ◇ *Dynamic problem features* outline the state of variable x_i with respect to the current node LP solution.
- ◇ *Dynamic optimization features* represent the overall statistical effect of variable x_i with respect to the optimization process.

At a given node of the B&B tree, the features vector ϕ_i represents the state of the candidate variable x_i. The SB score y_i, explicitly computed for the set of training instances, completes the pair (features, label). The learning task is a regression one, and it is performed with *Extremely Randomized Trees* (Geurts et al 2006), also known as *ExtraTrees*. ExtraTrees is an averaging ensemble method, based on Decision Trees, which can be employed for classification as well as regression. The purpose of averaging is that of reducing the variance of a prediction by combining many predictors. In particular, ExtraTrees randomly perturbs the construction of the regression trees, selecting a random subset of features and drawing a random pool of thresholds to determine best splits.

[1] Note that SB+PC and RB differ on the switch from SB to PC: at a certain fixed depth of the tree for the former method, whereas depending on each variable's reliability (i.e., past usage) for the latter.

A set of random, small-size problems is used for collecting SB-like examples, and the final training dataset includes 10^5 observations, i.e., pairs of features of a candidate variable at a given node and its SB score. A subset of instances coming from MIPLIB (Bixby et al 1996; Achterberg et al 2006) is employed for assessing the learned heuristic by comparing it with five concurrent ones: random branching, most-infeasible branching, non-chimerical branching, full strong branching and reliability branching. Experiments are performed with and without time and nodes limits.

In general, results are inferior to the state-of-the-art RB, but still showing that the learned branching efficiently imitates FSB. As the authors point out, the reduced time spent at each node allows the learned scheme to explore more nodes, which is obviously a key of success. Slightly better results can be obtained when tuning the learning to specific classes of problems, suggesting one of many possible research directions. Further details about the ML-based approximation of SB can be found in Marcos Alvarez (2016).

Two are the main differences between Marcos Alvarez et al (2017) and Marcos Alvarez, Wehenkel, and Louveaux (2016), the authors' other attempt in developing a learned branching strategy. The explored paradigm is that of *online* learning. In contrast with the offline upfront previously discussed, data is now generated and learned on-the-fly, within the B&B process itself. This implies that no preliminary and separated training phase is needed anymore. Still aiming at learning a fast approximation of SB, and keeping the same features of Marcos Alvarez et al (2017), the other novelty consists in the introduction of a reliability mechanism, very similar to the one in Achterberg et al (2005). More specifically, depending on the number of times a real SB score was already computed for a certain variable (the RB threshold $\eta = 8$ is used), one could deem the candidate reliable, and hence trust an approximate version of its SB score. Otherwise, if the variable is deemed unreliable (i.e., the information exploited about it is not enough to portray it correctly) a real SB score is computed. At a given node of the B&B tree, every time a variable is not deemed reliable, the features vector is computed together with the SB score, and a new example (features, label) can be added to the training set. The learning is performed with a simple linear regression, guided by a line search gradient descent algorithm (see, e.g., Nocedal and Wright 2006).

The defined *online learning* branching (olb) strategy exhibits at least one limitation, i.e., that of not adapting over time, with respect to possible changes of the variables dynamics along the B&B tree, as suggested by (iv). Indeed, at some point in the tree all variables would be deemed reliable, and the learning would stop updating. To fix the issue, the authors propose a *perpetual* version of olb (oplb). In short, the improved method allows the addition of new examples also when a variable is deemed reliable. Although the SB scores are not computed in the first place, features are stored for a reliable candidate at a given node; eventually, when both child nodes are explored, the SB information becomes readily available and a new pair (features, label) can be added to the dataset.

Both olb and oplb are compared on MIPLIB (Bixby et al 1996; Achterberg et al 2006) against three other heuristics: full strong branching, reliability branching and the learned branching of Marcos Alvarez et al (2017) (actually, the one of Marcos Al-

varez et al (2014), the preliminary version of the same paper). Results are interesting, in that both olb and oplb are competitive with RB in terms of nodes and time performance profiles. Again, method and results are further discussed in Marcos Alvarez (2016).

Khalil, Le Bodic, Song, Nemhauser, and Dilkina (2016) pursue the same goal of learning an effective approximation of SB. The authors aim at defining a method imitating SB in its node-efficiency, while being low-cost in terms of computational time and adaptive with respect to different instances to be solved. The scheme consists of three phases. First of all, during a *data collection* phase, SB is employed as variable branching rule up to a limited number of nodes $\theta(=500)$ of the B&B tree. The performed SB decisions are observed and registered as (features, label) pairs in the training dataset. Second, a *supervised ranking* task is executed, and a ranking function learned. Finally, the *ML-based B&B* takes over: the optimization continues employing the learned ranking function as branching heuristic, while SB is turned off.

The introduced ranking framework seems a natural approach for variable selection: predicting a ranking rather than a scalar score (as it is done by means of regression in Marcos Alvarez et al (2017) and Marcos Alvarez et al (2016)) is what a branching heuristic is ultimately pursuing. Note that the strategy resembles SB+PC and RB, in that SB is used only up to a certain point, i.e., while candidate variables are uninitialized or deemed unreliable. The outlined technique acts on-the-fly, without any expensive upfront, but does not adapt overtime, in contrast with the online perpetual approach (oplb) proposed in Marcos Alvarez et al (2016). In this sense, all procedures (learned of Marcos Alvarez et al (2017), olb of Marcos Alvarez et al (2016) and SB+ML of Khalil et al (2016)) seem to suffer the same limitation to adapt with respect to the tree evolution. The perpetual version oplb of Marcos Alvarez et al (2016) is the only method taking explicitly care of the adaptive issue. However, a little bit unexpectedly, it does not lead to significant improvements when compared to its halting counterpart.

Going back to Khalil et al (2016), features are divided into two categories:

- ◇ *Atomic features* describe the role of a candidate branching variable within a particular node of the tree. In particular, 72 atomic measures are designed (in a fashion similar to Marcos Alvarez et al (2017)), and are further split into *static* and *dynamic*. The former set includes those characteristics of the problem shared by the whole tree (they are computed at the root node), the latter encompasses the traits associated with a particular LP node.
- ◇ *Interaction features* consist of products of two static features. The whole features vector can be interpreted as a degree-2 polynomial kernel $K(\mathbf{u},\mathbf{v}) = (\mathbf{u}^T\mathbf{v}+1)^2$, acting in the 72-dimensional space of atomic features. More details on kernel mappings can be found in Bishop (2006).

Given the goal of learning a ranking function, instead of a regression one, while SB scores are computed for the training set of examples, they are not directly employed as labels. The proposed scheme is a binary labeling: labels are either 1 or 0, depending on their being or not in a fraction of top scoring variables at a given node.

More specifically, denoting by $\mathcal{C}_j$ the set of candidate variables at node N_j, the best SB score is $SB^j_* = \max_{i\in\mathcal{C}_j}\{SB^j_i\}$; a label for variable x_i at node N_j is computed as

$$y^j_i = \begin{cases} 1, & \text{if } SB^j_i \geq (1-\alpha)\cdot SB^j_* \\ 0, & \text{otherwise,} \end{cases} \tag{6}$$

where $\alpha \in [0,1]$ decides the portion of variables that are considered good, i.e., sufficiently close to the maximum score. This relaxed definition of "best" branching variables allows to take into account many good candidates in the learning. Moreover, as the authors point out, this scheme should prevent the execution of irrelevant learning, such as the correct relative ranking of variables with low SB scores.

The ranking formulation follows a *pairwise approach*: pairs of candidates are considered, and the objective is to rank them as SB does. Formally, a set of pairs $\mathcal{P}_j = \{(x_i,x_k) : i,k \in \mathcal{C}_j \text{ and } y^j_i > y^j_k\}$ is considered for every node N_j, and the learned ranking seeks to violate as few as possible *pairwise ordering* constraints of type

$$\forall j \in \{1,\dots,\theta\}, \forall (x_i,x_k) \in \mathcal{P}_j : y^j_i > y^j_k. \tag{7}$$

The Support Vector Machine (SVM) classification approach of Joachims (2006), SVM^{rank}, optimizes an upper bound on the number of violated constraints in (7), and it is used to approximate the ranking problem. The learned ranking function is then directly plugged in the branching system.

The ranking heuristic SB+ML is compared against other four strategies: CPLEX default of version 12.6.1 (in the spirit of hybrid branching, Achterberg and Berthold 2009), SB, PC and SB+PC. The comparison shows that SB+ML solves more instances than both PC and SB+PC, requiring fewer nodes. These savings counterbalance the more time spent per node of SB+ML, which could be imputable to features computation.

A step further is taken by Khalil (2016), who briefly explores the possibility of employing online and reinforcement learning in order to build a branching heuristic. The motivation of such an approach comes from the nature of B&B itself, which makes it possible to model the branching decision as a *multi-armed bandit* (MAB) problem (Robbins 1952). In a MAB problem, at each round an agent selects one of many available actions (arms) and registers the reward associated with the performed choice. The intuitive goal is to identify and follow a sequence of actions maximizing the long-term reward (or minimizing the regret). Note that the B&B system can easily be interpreted in MAB terms: every node corresponds to a round, and every candidate variable to an available arm. A (not fully discussed) performance measure can then be used as reward function to guide the agent in its selections.

A preliminary setup mentioned in Khalil (2016) outperforms on average PC. Although at its start, this seems a promising path to be explored.

Exploiting the framework of reinforcement learning (cf. Section 2.3) for variable branching, and more generally within the B&B technology appears suitable, given the inherent sequential nature of B&B. In particular, the idea that at a given node one should take into account future steps is already expressed in Glankwamdee and

Linderoth (2011). In that work, SB is interpreted as a greedy heuristic, optimizing the dual-bound improvement (a reward) as much as possible at the current node only. Though the two-steps lookahead strategy devised is costly, we should maybe take into consideration the sequentiality of the whole tree-process in future research.

We summarize the three main ML-based contributions discussed in this section in Table 1. For each work we report the chosen learning setting, details on the employed test setting (dataset, solver's specifications, compared algorithms and measures) and a brief descriptive summary of the results.[2]

Table 1 Synoptic comparison of the three discussed ML-based branching heuristics. For each work we report: learning setting, test set composition and specifications, employed solver and tested settings, list of compared algorithms (novel methods are in bold), measures of comparison, and a descriptive summary of the results.

	M.Alvarez et al (2017)	**M.Alvarez et al (2016)**	**Khalil et al (2016)**
Learning setting	ExtraTrees for regression (offline, supervised learning)	Linear regression (online and adaptive supervised learning)	SVM^{rank} (learning-to-rank with pairwise approach, on-the-fly supervised learning)
Test instances	. 150 random . 44 MIPLIB 3.0+2003 small to medium size	. 44 MIPLIB 3.0+2003 10 seeds small to medium size	. 84 MIPLIB 2010 10 seeds
Solver	CPLEX 12.2	CPLEX 12.6	CPLEX 12.6.1
Setting(s)	w/ and w/o: heuristics, cuts, presolve, timelimit 10min, nodelimit 10^5	disabled presolve, timelimit 2h	cutoff provided, timelimit 5h, disabled heuristics, cuts at root only
Algorithms	. random branching . MIB . NCB . FSB . RB . **learned**	. FSB . RB . learned . **olb** . **oplb**	. CPLEX default . SB . PC . SB+PC . **SB+ML**
Measures	. closed gap (within limits) . solved (within limits) . # nodes . time	. performance profiles (nodes and time)	. # unsolved . # nodes . time
Results summary	**learned** well imitates FSB. LP gap is improved w.r.t. FSB in the setup w/ limits, but RB dominates. Good also w/ cuts and heuristics in the setup w/o limits.	**olb** and **oplb** are competitive with RB in both performance profiles. The adaptive **oplb** does not significantly improve **olb**.	**SB+ML** solves more instances than PC and SB+PC. On average, it requires fewer nodes than PC and SB+PC, and slightly more than CPLEX default.

[2] Note that Marcos Alvarez et al (2016) follows Marcos Alvarez et al (2017), where 2017 is the year of journal publication of Marcos Alvarez et al (2014).

5 Searching the branching tree

As discussed, the search process of B&B highly relies on how the exploration of the decision tree is performed. The mechanism of implicit enumeration has two general goals:

(I) find quickly a good (possibly optimal) *integer feasible* solution;
(II) provide a *certificate of optimality* for the current incumbent, i.e., prove that no better solution exists.

Indeed, a solution of good quality helps the implicit enumeration as it allows to discard provably useless branches, limiting the number of explored nodes and focusing only on worthy subproblems. Thus, it appears clear that the order in which the nodes of the branching tree are selected for exploration has a significant impact on the efficiency of the B&B method.

Many heuristic rules for searching the B&B tree have been proposed in the years, with the aim of attaining one goal or another, or trying to balance both objective in some way. We will briefly discuss some of them in what follows. What is certain is that no single heuristic dominates the others, their performance very likely depending on the class of considered MILP problems.

Far from being confined to MILP research, the exploration of a decision tree is actually a very interdisciplinary theme, common in the AI community as well. As a consequence, we should not be too surprised to find out that the recent works applying ML techniques to the B&B tree search come from AI researchers. Similarly to how we discussed branching in Section 4, we will start in Section 5.1 by retrieving early optimization works somehow anticipating the development of adaptive and informed search heuristics. We will then present in Section 5.2 two recent attempts in this direction, employing different learning frameworks.

5.1 Preliminary considerations on search

We begin our agenda with the work of Linderoth and Savelsbergh (1999), which stands as a survey of B&B search heuristics as well as a baseline ground for a general discussion on the topic. In particular, the authors compare 13 different node selection methods, identifying three leading evaluation measures to rank them, in line with goals (I) and (II) above. The ranking is performed with respect to (in order of importance):

- ◇ value of the best solution obtained,
- ◇ provable optimality gap, and
- ◇ computating time.

The various heuristics are categorized into four different classes. Among *static methods*, depth-first and best-first searching paradigms are interpreted as extreme points of view on the node selection task: the former is associated with goal (I), the latter is linked to (II). This understanding naturally motivates *two-phase methods*, alternating depth-first and best-first in order to balance the search objectives. Further, the idea of

making more intelligent selections in order to find new improved incumbent solutions is at the root of *estimate-based methods*. Criteria such as best-projection and best-estimate exploit the notion of estimating the value of the best feasible solution with respect to a certain subtree. Finally, *Backtracking methods* also use estimates to guide the search, with the aim of avoiding superfluous nodes. As expected, such estimation rules become more realistic as they get more complex, involving many different characteristics (or features) such as pseudocosts, fractionalities and probabilities of successful rounding.

As the authors point out, given that the effectiveness of the above methods depends on the problem type, one would seek a search strategy that could adapt to different instances. In particular, some grade of adaptiveness could be pursued in the combination of best-first and depth-first strategies, which, borrowing a reinforcement learning vocabulary (see Section 2.3), corresponds to balancing exploration and exploitation in the B&B environment. Overall, it seems that the systematic analysis performed (for the first, and still most complete, time) in Linderoth and Savelsbergh (1999) can now be revisited in the light of modern ML techniques.

A different observation on the nature of the B&B tree underpins the work of Fischetti and Monaci (2014). The authors support the claim that high-sensitivity and erraticism are inherent properties of tree search, due to the very same exponential nature of the enumeration tree, which ought to be exploited in a beneficial way. Their *bet-and-run* approach first triggers randomization, producing few short runs. Among those different runs, a bet is made on the most favorable one, which is then alone brought to completion.

In more details, the algorithm makes use of a restart policy reminiscent of Karzan et al (2009) and Fischetti and Monaci (2012a). Within a *sampling phase*, $C = 5$ random clones of the problem are created and solved up to $N = 5$ nodes only. The best clone with respect to some aggregated indicator is selected and fully solved in the *long run*. Two are the key aspects of the method.

1. First, one needs to generate meaningful diversity while randomizing, without degrading the average performance. Moreover, the randomization should happen after the preprocessing and the solution of the root node, in order to limit the computational overhead. The implemented strategy consists of temporary replacing the objective function of a clone with a random one, having fixed all nonbasic variable with nonzero reduced cost. Reoptimizing will lead to a different basis on the optimal face of the LP relaxation of the initial MILP, from which to start the search. A cap on the number of performed simplex pivots is enforced in order to maintain a short computing time.
2. Second, a selection rule for the "best" clone run must be defined, and hence one needs to identify some measures evaluating the performed (short) searches. As the authors recognize, erraticism itself precludes the definition of a perfect criterion. Hence, the aim is that of establishing a positive correlation between the clone to be selected and the *a posteriori* better run. Note that this notion naturally calls for a supervised learning framework, the *a posteriori* best run consisting in an example's label.

The authors identify 8 indicators of performance, with priority order, extracting information about: open nodes, lower bound improvements, infeasibilities and number of simplex iterations. The proposed evaluation scheme discards the indicators that do not provide discriminant information, and breaks ties to favor the first (default and unperturbed) clone. Note that the defined criterion bases its decision-making on the very beginning of the search, only.

The bet-and-run algorithm is compared with the default CPLEX setting (no dynamic search) and two *a posteriori* oracle algorithms, on 344 instances from MIPLIB 2010 (Koch et al 2011) and COR@L (2017). The algorithm produces savings in terms of time and nodes for medium to hard instances, but the computational overhead does not payoff for easy instances. A modified version, denoted as *hybrid*, is tested, which prescribes to run CPLEX default for NR nodes, to understand whether the instance at hand is "easy". If yes, the problem is solved by CPLEX with no modifications. Otherwise, perform the sampling phase of bet-and-run: if the selected clone is the unperturbed one, continue with default; else, continue with bet-and-run selection for the long run. The hybrid variation is beneficial on average, although it does not seem to fully solve the overhead issue.

Interestingly, throughout the paper, the authors themselves point out some possibilities for improvement that could involve the use of ML tools.

- Learning algorithms could provide a more sophisticated decision rule for the best clone selection criterion. A classification mechanism could improve the selection.
- While $NR = 500, 1000$ are tested, the computation of the parameter could be adaptive and performed on-the-fly, i.e., estimating the hardness of the remaining tree or, equivalently, the hardness of instance.
- The restart strategy could be refined by leveraging information of past runs.

We are now going to see how the main axes of these analyses on B&B search will be reinterpreted in a learning framework.

5.2 Learning approaches to B&B search

Keeping in mind the recognized needs and goals of a search strategy for B&B, we will present in this section two new approaches for the topic. The tools employed in those attempts may appear uncommon to a MILP practitioner, and we will try to outline their potentials as well as their limitations.

In Sabharwal, Samulowitz, and Reddy (2012) the exploited framework is that of reinforcement learning (see Section 2.3). A multi-armed bandit (see Section 2.3) structure is proposed for MILP search, in the form of a modified version of the Upper Confidence bounds for Trees (UCT) (Kocsis and Szepesvári 2006) technique. Namely, UCT is a method for Monte Carlo Tree Search balancing exploration and exploitation, and it is based on the selection strategy of Upper Confidence Bounds (UCB1), which was introduced in Auer et al (2002). In a nutshell, UCT works on an underlying tree T and consists of two alternating phases. Within the *node selection phase*, T is traversed from its root to a leaf node: at each node N the rule is to move to the child

N' with higher *UCT score*; ties are arbitrarily broken. Once a leaf L is reached, a *tree update phase* is performed: an updated score is computed for L and it is propagated upwards to the root, following the outbound path in reverse order and adjusting the estimates for the encountered nodes.

The authors employ a simplified version of the UCT score of a node N (the $\varepsilon-$greedy version in Auer et al 2002), consisting of a balanced sum of two terms:

$$\text{score}(N) = \text{estimate}(N) + \Gamma \cdot \frac{\text{visits}(P)/100}{\text{visits}(N)}, \tag{8}$$

where estimate(N) is some measure of quality of node N, P is the parent node of N and visits($\cdot$) counts the number of times a node has been already visited by the search algorithm. The parameter Γ controls the balance between exploration (second term) and exploitation (first term): nodes with high estimate are pursued, but other nodes get priority if they have been visited only a fraction of the times compared to their siblings. In the original context of UCT (adversary game tree search), the estimate($\cdot$) values are initialized by *random playouts*: the game is played many times until its very end by selecting casual moves, each play yielding a certain result that is used as measure of quality for the traveled path. Moreover, the tree update phase is carried out by a so-called *backup operator*, which usually assigns to a node N in the path from leaf to root the average of the values seen in the subtree rooted at N.

Given the differences between the original context of UCT and that of MILP, the algorithm must be appropriately modified. The goal of such adjusted UCT is that of guiding B&B search by expanding open nodes as UCT would expand them.

For a start, B&B tree search is a single-agent process, and it is clear that the MILP framework cannot afford random playout samplings for initialization purposes. Instead, the fact that branching provides guaranteed LP bounds (a strong heuristic) is exploited, so that the quality estimates consist of normalized LP objective values. Having guaranteed bounds requires changes be made for the backup operator as well: instead of an averaging one, a *max-style* updating rule seems more suited. In short, each node's estimate is updated with the maximum between the estimates of its children nodes, so that at any N, estimate(N) equals the best objective value seen in the subtree rooted at N. Note that when a node is closed by B&B (i.e., fathomed), the search will not have any reason to visit it again. In this sense, exploitation is not always meaningful in this setting, and subtrees with no open nodes left should be disregarded by future UCT searches.

The UCT-based search strategy is compared with three others, namely, best-first, (graph-theory) breadth-first and CPLEX default heuristic. A general MILP solver performs B&B by internally maintaining the list of open nodes, but in order to apply the UCT-based method one needs to maintain an underlying entire tree structure to guide the search, i.e., nodes already explored are not removed. This additional architecture introduces significant overhead. To limit it, and supporting the notion that decisions are more crucial at the top levels of the tree, each tested strategy is performed on the first 128 nodes only, then switching to the CPLEX default one. A total of 170 instances from MIPLIB Koch et al (2011) are used, with 600 seconds time limit, and the balancing parameter is tuned to $\Gamma = 0.7$. The geometric means of runtime, num-

ber of searched nodes and of simplex iterations are all improved by the UCT-based technique.

Some remarks before moving on. Note that the essential part of the reward measure consists of LP objective values, which is combined with the visits counter. Considering this, it seems that UCT improvements are gained thanks to a balanced usage of best-first and breadth-search-like schemes, without the exploitation of other state information. Moreover, the original UCT algorithm as in Kocsis and Szepesvári (2006) treats every internal node as a different MAB problem, a property which the authors do not consider sustainable in the MILP context. However, detaching for a moment from architectural issues, we could easily imagine bandit problems at every node of the B&B tree, eventually dealing with branching decisions, as proposed in Khalil (2016). An interesting question is whether the two processes of variable and node selection could be unified under the MAB scheme, or, more generally, within a (reinforcement?) learning framework.

Learn an adaptive and problem-specific search strategy is the goal of He, Daume III, and Eisner (2014). This work makes use of *imitation learning* (or *behavioral cloning*, see Sammut 2011), a paradigm very common in robotics. In general, imitative methods involve an expert performing some task, and having its actions recorder together with a description of the current situation. A dataset of (situation, action) pairs is given as input to a learning program, which then produces as output a set of rules (i.e., policies) reproducing the expert behavior with respect to the performed task.

In the context of B&B, the expert part is played by a MILP solver and of a simple defined *oracle*. While it would be ideal to have an oracle expanding an optimal sequence of nodes and fulfilling (I) and (II) (and hence also (i), i.e., minimizing the number of explored nodes), the designed oracle only cares of (I). Namely, the explicitly declared goal is that of finding good (possibly optimal) solutions quickly, but without providing a certificate of optimality. This modeling choice is motivated by the wish of allowing a more aggressive pruning of the tree branches, and motivated by the idea that it may be possible in the future to reach a "user-specified trade-off between solution quality and searching time". Indeed, it is worth observing that the proposed framework does not guarantee an optimality certificate because it prunes subtrees potentially containing the optimal solution (besides the training phase in which the optimal solution is known).

More in details, the assumption is that optimal solutions of training problems are given. The oracle node selection rule π_S^* will always pursue the branches containing the optimal solution. Nodes expanded in this process are called *optimal nodes*; the non-optimal nodes are pruned from the tree by the oracle pruning rule π_P^*. The B&B is framed as a sequential process within the state space $\mathcal{S}$ consisting of the visited nodes and their LP bounds. Two policies that should guide the search are learned as rules. Namely,

- The (learned) *node selection policy* π_S prescribes which node should be expanded next. Namely, π_S provides a priority order for the queue of open nodes, deciding which one should be popped. The action space of π_S is $\{$*select node* $N_i : N_i \in$ *queue of open nodes*$\}$.

◇ The (learned) *node pruning policy* π_P determines whether a popped node is worth to be expanded. Note that in this context the terms "pruning" and "fathoming" are not interchangeable: a node that is not fathomed by infeasibility, bound or integrality could still be pruned because of its non-optimality. The action space of π_P is $\{prune,\ expand\}$, so that π_P actually behaves like a binary classifier.

In fact, not only π_P, but the whole imitation problem can be reduced to supervised learning, as shown in Syed and Schapire (2010). The oracle actions a_t^* can be interpreted as predictions with respect to a features vector description of state s_t. Instead of forecasting a regression score, π_S itself can be framed as a classifier of pairs, for the problem where one aims at learning a ranking of the open nodes in queue. The procedure is in the spirit of that employed by Khalil et al (2016) for variable selection purposes.

Features are divided into three categories.

◇ *Node features* include bounds and objective function estimation at a given node, together with indications about the current depth and the (parental) relationship with respect to the last processed node.
◇ *Branching features* describe the variable whose branching led to a given node, in terms of pseudocosts, variable's value modifications and bound improvement.
◇ *Tree features* consider measures such as the number of solutions found, global bounds and gap, with respect to the computed B&B tree as a whole.

Two different feature maps are defined for the policies: while π_S bases its predictions mainly on node and branching features, π_P employs mostly branching and tree features. All features are combined with the depth of the measured node by means of partitioning the tree in 10 uniform levels, and are appropriately normalized with respect to the root node's values. The designed characteristics do not constitute a computational burden, being easily obtainable from a MILP solver.

In the experiments, instances are borrowed from four diverse libraries. The MILP solver SCIP (2017) is run to optimality and the delivered optimal solution is used within the oracle to initialize the training phase. The training of the policies is performed iteratively on problem classes. Every iteration provides updated information and collects new examples during the B&B runs, taking care of correctly ranking a node when it enters the open nodes queue and pruning it if non-optimal, after it is extracted from the queue. As already said, the learning of π_S and π_P is in fact a classification task, and attention is paid to properly tune parameters.

At test phase, the resulting algorithm, called *DAgger*, is compared with SCIP and Gurobi (2017), taking into account the trade-off between runtime and solution quality in the comparison. The adaptive solver performs well on different classes of problems, and it seems to fulfill the three-level adaptiveness sought by the authors, with respect to: (1) problem type, (2) specific instance and (3) different stages of the B&B optimization.

Further analysis shows the pruning policy having more impact, possibly due to the fact that other heuristic components interfere with node selection. Not so surprising are the findings of the features analysis. In general terms, π_S imitates depth-first search at the top of the tree, but also considers historical estimates in lower levels. The

branching variable's measures seem to affect π_P, which keeps the pruning cautious when only few solutions are known.

Note that the branching decision is explicitly taken into account in the design of the search strategy by the second group of features, reflecting the idea that these two heuristic processes should be intertwined within the learning as they indeed are in the optimization. Moreover, the authors claim that the design of DAgger takes into account the complex sequential nature of B&B by modeling the influence of actions over future states, a task that could not be performed by standard supervised learning. However, the work is based on the assumption that current solvers' strategies are the "experts" to be imitated, and drops the seek of certified optimality for speed.

We summarize the two main ML-based contributions discussed in this section in Table 5.2. Again, for each work we report the chosen learning setting, details on the employed test setting (dataset, solver's specifications, compared algorithms and measures) and a brief descriptive summary of the results.

Table 2 Synoptic comparison of the two discussed learning approaches to B&B search. For each work we report: learning setting, test set composition and specifications, employed solver and tested settings, list of compared algorithms (novel methods are in bold), measures of comparison, and a descriptive summary of the results.

	Sabharwal et al (2012)	**He et al (2014)**
Learning setting	UCT (reinforcement learning)	Imitation learning
Test instances	. 179 various benchmarks	. 36 MIK . 120 Regions . 40 Hybrid . 300 CORL@T
Solver(s)	CPLEX 12.3	. SCIP 3.1.0 (CPLEX 12.6 for LP) . Gurobi 5.6.2
Setting(s)	node and branch callbacks on, 600s timelimit	average runtime and # of nodes of the proposed B&B are used as timelimit for SCIP and nodelimit for Gurobi, respectively
Algorithms	. **UCT** . CPLEX default . best-first . breadth-first	. $\pi_{\mathbf{S}} + \pi_{\mathbf{P}}$ (selection + pruning) . $\pi_{\mathbf{P}}$ (pruning policy only) . SCIP (time) . Gurobi (node)
Measures	. runtime . # nodes . # simplex iterations	. speedup w.r.t. SCIP default . optimality gap . integrality gap
Results summary	**UCT**-based technique improves the geometric means of all considered measures.	Good adaptive performance on all classes of problems: $\pi_{\mathbf{P}}$ seems to have more impact, $\pi_{\mathbf{S}}$ likely interferes with solver's other components.

6 Overview and conclusions

In this paper, we have surveyed learning techniques to deal with the two most crucial decisions in the branch-and-bound algorithm for MILP, namely variable and node selections. Because of the lack of deep mathematical understanding on those decisions, the classical and vast literature in the field is inherently based on computational studies and heuristic, often problem-specific, strategies. Although our survey is mostly concerned with the recent methods that explicitly consider (machine) learning techniques, we have taken the perspective of interpreting some of the previous fundamental contributions in the light of those techniques, so as to give a more complete overview and to possibly outline new points of view.

It is worth observing that we have not touched in our discussion a nowadays fundamental component of branch-and-bound algorithms and codes for MILP, namely parallelization. Modern MILP solvers are developed, tested and used within multi-thread computing environments and more and more research is devoted to improve in the use of multi-threading. Although the papers we surveyed almost never discuss the issue, it is not difficult to imagine the use of the learning algorithms for both variable and node selections in a parallel environment. However, evil is in the details and the discussed paradigms have to be treated / extended with care.

Of course, variable and node selections are not the only important decisions in enumerative algorithms in general. One of the areas in which modern learning techniques could result crucial is that of predicting the difficulty of an instance, for example by taking into account the size and the shape of the enumeration tree. The problem of hardness prediction is not new. Since the first estimation of efficiency for backtracking methods of Knuth (1975), the question has been a common interest of the optimization and the ML communities, which developed their own algorithms in the past decades. Indeed, the practical impact of such a prediction is wide and important, especially given the time and resources limits that one has to confront when dealing with hard problems. Discussing this topic in details is outside of the scope of the present paper. However, again following the pattern of interpreting some old(er) contributions in the light of modern learning algorithms and then considering the most recent works, we refer the interested reader to Cornuéjols, Karamanov, and Li (2006) for the former and to Hutter, Xu, Hoos, and Leyton-Brown (2014) and Marcos Alvarez, Wehenkel, and Louveaux (2015) for the latter.

Acknowledgments

We like to thank Yoshua Bengio for his support in our learning curve. Additional thanks go to Laurent Charlin, Mathieu Tanneau, Claudio Sole and François Laviolette for interesting discussions on the topic.

References

Abdi H, Williams LJ (2010) Principal component analysis. Wiley Interdiscip Rev Comput Stat 2:433–459

Achterberg T, Berthold T (2009) Hybrid Branching. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Springer, Berlin, Heidelberg, pp 309–311

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

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

Ansótegui C, Sellmann M, Tierney K (2009) A gender-based genetic algorithm for the automatic configuration of algorithms. In: CP, Springer, Berlin, Heidelberg, pp 142–157, DOI 10.1007/978-3-642-04244-7_14

Applegate D, Bixby R, Chvátal V, Cook W (2007) The Traveling Salesman Problem. A Computational Study. Princeton University Press

Auer P, Cesa-Bianchi N, Fischer P (2002) Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2-3):235–256, DOI 10.1023/A:1013689704352

Bellman R (1961) Adaptive Control Processes. Princeton University Press

Benichou M, Gauthier J, Girodet P, Hentges G (1971) Experiments in mixed-integer programming. Mathematical Programming 1:76–94

Bertsekas DP, Tsitsiklis JN (1996) Neuro-Dynamic Programming, 1st edn. Athena Scientific

Bishop CM (2006) Pattern Recognition and Machine Learning. Information Science and Statistics, Springer-Verlag New York, Inc.

Bixby R, Ceria S, McZeal C, Savelsbergh M (1996) An updated mixed integer programming library: Miplib 3.0

COR@L (2017) Computational Optimization Research at Lehigh. URL `https://coral.ise.lehigh.edu`

Cornuéjols G, Karamanov M, Li Y (2006) Early estimates of the size of branch-and-bound trees. INFORMS Journal on Computing 18(1):86–96, DOI 10.1287/ijoc.1040.0107

CPLEX (2017) URL `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html`

Domingos P (2012) A few useful things to know about machine learning. Commun ACM 55(10):78–87, DOI 10.1145/2347736.2347755

Fischetti M, Monaci M (2012a) Backdoor branching. INFORMS Journal on Computing 25(4):693–700, DOI 10.1287/ijoc.1120.0531

Fischetti M, Monaci M (2012b) Branching on nonchimerical fractionalities. Operations Research Letters 40(3):159–164

Fischetti M, Monaci M (2014) Exploiting erraticism in search. Oper Res 62(1):114–122

Fischetti M, Lodi A, Monaci M, Salvagnin D, Tramontani A (2016) Improving branch-and-cut performance by random sampling. Mathematical Programming Computation 8(1):113–132

Geurts P, Ernst D, Wehenkel L (2006) Extremely randomized trees. Mach Learn 63(1):3–42, DOI 10.1007/s10994-006-6226-1

Gilpin A, Sandholm T (2011) Information-theoretic approaches to branching in search. Discrete Optimization 8(2):147–159, DOI 10.1016/j.disopt.2010.07.001

Glankwamdee W, Linderoth J (2011) Lookahead branching for mixed integer programming. In: Twelfth INFORMS Computing Society Meeting, INFORMS, pp 130–150

Gomory R (1960) An algorithm for the mixed integer problem. Tech. Rep. RM-2597, The Rand Corporation

Goodfellow I, Bengio Y, Courville A (2016) Deep Learning. MIT Press, URL `http://www.deeplearningbook.org`

Gurobi (2017) URL `http://www.gurobi.com`

Hamerly G, Elkan C (2003) Learning the k in k-means. In: NIPS, vol 3, pp 281–288

Hastie T, Tibshirani R, Friedman J (2009) The Elements of Statistical Learning: Data Mining, Inference and Prediction, 2nd edn. Springer series in statistics, Springer

He H, Daume III H, Eisner JM (2014) Learning to search in branch and bound algorithms. In: Ghahramani Z, Welling M, Cortes C, Lawrence ND, Weinberger KQ (eds) Advances in Neural Information Processing Systems 27, Curran Associates, Inc., pp 3293–3301

Hutter F, Xu L, Hoos HH, Leyton-Brown K (2014) Algorithm runtime prediction: Methods & evaluation. Artificial Intelligence 206:79–111, DOI 10.1016/j.artint.2013.10.003

Joachims T (2006) Training linear SVMs in linear time. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, pp 217–226

Kadioglu S, Malitsky Y, Sellmann M (2012) Non-model-based search guidance for set partitioning problems. In: AAAI

Karzan FK, Nemhauser GL, Savelsbergh MWP (2009) Information-based branching schemes for binary linear mixed integer problems. Math Prog Comp 1(4):249–293, DOI 10.1007/s12532-009-0009-1

Khalil E (2016) Machine learning for integer programming. In: Proceedings of the Doctoral Consortium at the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)

Khalil E, Le Bodic P, Song L, Nemhauser G, Dilkina B (2016) Learning to branch in mixed integer programming. In: AAAI

Knuth DE (1975) Estimating the efficiency of backtrack programs. Math Comput 29(129):122–136

Koch T, Achterberg T, Andersen E, Bastert O, Berthold T, Bixby R, Danna E, Gamrath G, Gleixner A, Heinz S, Lodi A, Mittelmann H, Ralphs T, Salvagnin D, Steffy D, Wolter K (2011) MIPLIB 2010. Mathematical Programming Computation pp 103–163

Kocsis L, Szepesvári C (2006) Bandit based monte-carlo planning. In: Machine Learning: ECML 2006, Springer, Berlin, Heidelberg, pp 282–293, DOI 10.1007/11871842_29

Land A, Doig A (1960) An automatic method of solving discrete programming problems. Econometrica 28:497–520

Liberto GD, Kadioglu S, Leo K, Malitsky Y (2016) DASH: Dynamic approach for switching heuristics. European Journal of Operational Research 248(3):943–953, DOI 10.1016/j.ejor.2015.08.018

Linderoth JT, Lodi A (2010) Milp software. Wiley encyclopedia of operations research and management science

Linderoth JT, Savelsbergh MWP (1999) A computational study of search strategies for mixed integer programming. INFORMS Journal on Computing 11(2):173–187, DOI 10.1287/ijoc.11.2.173

Lodi A (2010) Mixed integer programming computation. In: 50 Years of Integer Programming 1958-2008, Springer Berlin Heidelberg, pp 619–645

Lodi A (2013) The heuristic (dark) side of MIP solvers. In: Talbi EG (ed) Hybrid Metaheuristics, no. 434 in Studies in Computational Intelligence, Springer Berlin Heidelberg, pp 273–284

Lodi A, Tramontani A (2013) Performance variability in mixed-integer programming. In: Topaloglu H (ed) Tutorials in Operations Research, INFORMS, pp 1–12

MacQueen J (1967) Some methods for classification and analysis of multivariate observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, University of California Press, vol 1, pp 281–297

Marcos Alvarez A (2016) Computational and theoretical synergies between linear optimization and supervised machine learning. PhD thesis, Université de Liège, Liège, Belgique

Marcos Alvarez A, Louveaux Q, Wehenkel L (2014) A supervised machine learning approach to variable branching in branch-and-bound. Tech. rep., Université de Liège, URL `http://hdl.handle.net/2268/167559`

Marcos Alvarez A, Wehenkel L, Louveaux Q (2015) Machine learning to balance the load in parallel branch-and-bound. Tech. rep., Université de Liège, URL `http://hdl.handle.net/2268/181086`

Marcos Alvarez A, Wehenkel L, Louveaux Q (2016) Online learning for strong branching approximation in branch-and-bound. Tech. rep., Université de Liège, URL `http://hdl.handle.net/2268/192361`

Marcos Alvarez A, Louveaux Q, Wehenkel L (2017) A machine learning-based approximation of strong branching. Inf J Comput DOI 10.1287/ijoc.2016.0723

Nocedal J, Wright S (2006) Numerical Optimization, 2nd edn. Springer, New York

Padberg M, Rinaldi G (1991) A branch and cut algorithm for the resolution of large-scale symmetric traveling salesmen problems. SIAM Review 33:60–100

Robbins H (1952) Some aspects of the sequential design of experiments. Bull Amer Math Soc 58(5):527–535

Sabharwal A, Samulowitz H, Reddy C (2012) Guiding combinatorial optimization with UCT. In: Beldiceanu N, Jussien N, Pinson É (eds) Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, Springer Berlin Heidelberg, Lecture Notes in Computer Science, pp 356–361, DOI 10.1007/978-3-642-29828-8_23

Sammut C (2011) Behavioral cloning. In: Sammut C, Webb GI (eds) Encyclopedia of Machine Learning, Springer US, pp 93–97, DOI 10.1007/978-0-387-30164-8_69

SCIP (2017) URL `http://scip.zib.de/`

Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27:379–423 and 623–656

Sutton RS, Barto AG (1998) Reinforcement learning: An introduction. MIT press

Syed U, Schapire RE (2010) A reduction from apprenticeship learning to classification. In: Lafferty JD, Williams CKI, Shawe-Taylor J, Zemel RS, Culotta A (eds) Advances in Neural Information Processing Systems 23, Curran Associates, Inc., pp 2253–2261

Szepesvari C (2010) Algorithms for Reinforcement Learning. Morgan and Claypool